Von der Carl-Friedrich-Gauß-Fakultät
Technische Universität Carolo-Wilhelmina zu Braunschweig

ALGORITHMS FOR PACKING PROBLEMS

Zur Erlangung des Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation
von
Dipl.-Math. Oec. Nils Schweer
geboren am 02.09.1980
in Lengerich.

Eingereicht am: 01.02.2010
Mündliche Prüfung am: 19.03.2010
Referent: Prof. Dr. Sándor P. Fekete
Korreferent: Prof. Dr. Sven O. Krumke (2010)

Abstract

Packing problems belong to the most frequently studied problems in combinatorial optimization. Mainly, the task is to pack a set of small objects into a large container. These kinds of problems, though easy to state, are usually hard to solve. An additional challenge arises, if the set of objects is not completely known beforehand, meaning that an object has to be packed before the next one becomes available. These problems are called *online* problems. If the set of objects is completely known, they are called *offline* problems. In this work, we study two online and one offline packing problem. We present algorithms that either compute an optimal or a provably good solution:

Maintaining Arrays of Contiguous Objects. The problem of maintaining a set of contiguous objects (*blocks*) inside an array is closely related to *storage allocation*. Blocks are inserted into the array, stay there for some (unknown) duration, and are then removed from the array. After inserting a block, the next block becomes available. Blocks can be moved inside the array to create free space for further insertions. Our goals are to minimize the time until the last block is removed from the array (the *makespan*) and the costs for the block moves. We present inapproximability results, an algorithm that achieves an optimal makespan, an algorithm that uses only $O(1)$ block moves per insertion and deletion, and provide computational experiments.

Online Square Packing. In the classical *online strip packing problem*, one has to find a non-overlapping placement for a set of objects (squares in our setting) inside a semi-infinite strip, minimizing the height of the occupied area. We study this problem under two additional constraints: Each square has to be packed on top of another square or on the bottom of the strip. Moreover, there has to be a collision-free path from the top of the strip to the square's final position. We present two algorithms that achieve *asymptotic competitive factors* of 3.5 and 2.6154, respectively.

Point Sets with Minimum Average Distance. A *grid point* is a point in the plane with integer coordinates. We present an algorithm that selects a set of grid points (*town*) such that the average L_1 distance between all pairs of points is minimized. Moreover, we consider the problem of choosing point sets (*cities*) inside a given square such that—again—the interior distances are minimized. We present a 5.3827-*approximation algorithm* for this problem.

Zusammenfassung

Packprobleme gehören zu den am häufigsten untersuchten Problemen in der kombinatorischen Optimierung. Grundsätzlich besteht die Aufgabe darin, eine Menge von kleinen Objekten in einen größeren Container zu packen. Probleme dieser Art können meistens nur mit hohem Aufwand gelöst werden. Zusätzliche Schwierigkeiten treten auf, wenn die Menge der zu packenden Objekte zu Beginn nicht vollständig bekannt ist, d.h. dass das nächste Objekt erst verfügbar wird, wenn das vorherige gepackt ist. Solche Probleme werden *online* Probleme genannt. Wenn alle Objekte bekannt sind, spricht man von einem *offline* Problem. In dieser Arbeit stellen wir zwei online Packprobleme und ein offline Packproblem vor und entwickeln Algorithmen, die die Probleme entweder optimal oder aber mit einer beweisbaren Güte lösen:

Verwaltung von kontinuierlichen Objekten. Das Problem eine Menge von kontinuierlichen Objekten (Blöcke) in einem Array *möglichst gut* zu verwalten, ist eng verwandt mit Problemen der Speicherverwaltung. Blöcke werden in einen kontinuierlichen Bereich des Arrays eingefügt und nach einer (unbekannten) Dauer wieder entfernt. Dabei ist immer nur der nächste einzufügende Block bekannt. Um Freiraum für weitere Blöcke zu schaffen, dürfen Blöcke innerhalb des Arrays verschoben werden. Ziel ist es, die Zeit bis der letzte Block entfernt wird (*Makespan*) und die Kosten für die Verschiebe-Operationen zu minimieren. Wir geben eine komplexitätstheoretische Einordnung dieses Problems, stellen einen Algorithmus vor, der einen optimalen Makespan bestimmt, einen der $O(1)$ Verschiebe-Operationen benötigt und evaluieren verschiedene Algorithmen experimentell.

Online-Strip-Packing. Im klassischen *Online-Strip-Packing-Problem* wird eine Menge von Objekten (hier: Quadrate) in einen Streifen (unendlicher Höhe) platziert, so dass die Höhe der benutzten Fläche möglichst gering ist. Wir betrachten einen Spezialfall, bei dem zwei zusätzliche Bedingungen gelten: Quadrate müssen auf anderen Quadraten oder auf dem Boden des Streifens platziert werden und die endgültige Position muss auf einem kollisionfreien Weg erreichbar sein. Es werden zwei Algorithmen mit Güten von 3,5 bzw. 2,6154 vorgestellt.

Punktmengen mit minimalem Durchschnittsabstand. Ein Gitterpunkt ist ein Punkt in der Ebene mit ganzzahligen Koordinaten. Wir stellen einen Algorithmus vor, der eine Anzahl von Punkten aus der Menge aller Gitterpunkte auswählt, so dass deren durchschnittlicher L_1-Abstand minimal ist. Außerdem betrachten wir das Problem, mehrere Punktmengen mit minimalem Durchschnittsabstand innerhalb eines gegebenen Quadrates auszuwählen. Wir stellen einen 5,3827-Approximationsalgorithmus für dieses Problem vor.

Acknowledgments

First of all, I am very grateful to **Sándor Fekete**, my PhD advisor, for plenty of valuable thoughts and advice and to **Sven Krumke** for being the second referee of this thesis.

My special thanks go to **Tom Kamphans** for many helpful discussions and bright ideas and, of course, for espresso and cookies, and to **Christiane Schmidt** for spending time to listen to my ideas and providing helpful comments.

Many thanks to all my colleagues in the *Algorithms Group* for a pleasant working atmosphere, especially to **Tobias Baumgartner**, **Chris Gray**, and **Björn Hendriks** for proofreading.

I would like to thank **Erik Demaine**, **Günther Rote**, **Daria Schymura**, and **Mariano Zelke**—it was a pleasure to work with you on the paper that lead to Section 5.2.

Last but not least, I am very grateful to my family for supporting and believing in me all the time.

to my wife

Contents

Chapter 1

Introduction

Packing a set of objects into a container or into a set of containers has many applications in everyday life. Some of them are immediately evident, such as packing clothes into a suitcase before going on vacation or loading containers onto a ship; see Fig. 1.1. Some of them appear more hidden, *e.g.,* in *scheduling problems* where the task is to assign a set of jobs to a number of machines. If we model the machines as containers and the jobs as the objects which have to be packed, then minimizing the number of used containers, implies a schedule that uses a minimum number of machines.

Over the years, a huge variety of packing problems has been studied. They differ in the shape of the objects or in the shape of the container(s). Moreover, there are many additional constraints such as constraints on the order in which the objects have to be packed or the placement of the objects inside the container, *e.g.,* rotating objects might be allowed or not.

Popular packing problems are, *e.g.,* the *bin packing problem* and the *strip packing problem*. In the bin packing problem a set of one-dimensional objects with size less than one have to be packed into unit-sized containers. The task is to use as few containers as possible. The strip packing problem asks for a placement of rectangles inside a semi-infinite strip of width one, minimizing the height used. These problems have been studied in one, two, and three dimensions.

In the strip packing problem the container and the objects usually have a rectangular shape. If the objects are not rectangles but some (arbitrary) polygons, the challenge is even harder. The same holds if the objects have a regular shape but not the container. This is often the case in packing problems that arise in real world applications. A packing problem that appears in industrial applications is the computation of the volume of a trunk. Accord-

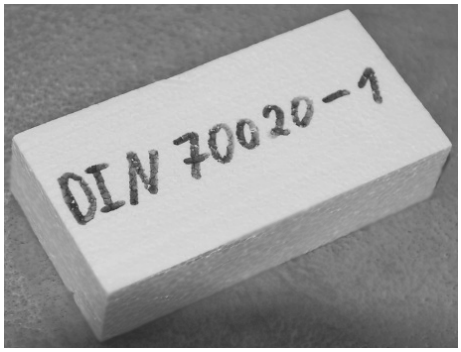

Figure 1.1: A box with volume one liter, that is used to measure the volume of a trunk (image source: `www.autobild.de`). A ship packed with containers (image source: `www.wikipedia.org`).

ing to the *Deutsches Institut für Normung*[1] *(DIN 70020-1),* the trunk volume is not the continuous volume (*e.g.,* the amount of water that can be filled into it) but rather the number of boxes with a volume of 1 liter that can be packed into it; see Fig. 1.1. A trunk is a three-dimensional—not necessarily convex—polygon. Thus, the task is to pack as many rectangular boxes as possible into a polygon. This problem has, *e.g.,* been studied in [Rei06].

Most of the packing problems are computationally *hard*, meaning that—roughly speaking—an optimal solution can only be found with enormous computational effort. However, for small instances exact solutions can be found in reasonable time. There are three main branches in the study of packing problems: exact algorithms, heuristics, and approximation algorithms.

Exact algorithms for packing problems are often based on methods that enumerate the solution space completely, *e.g.,* the *branch-and-bound method*. These approaches can be accelerated by providing good lower bounds on the solution value. For example, for the bin packing problem, *dual-feasible functions* [LMM02] often quickly provide near-optimal lower bounds. For a survey on exact methods see [FS98]. This survey also contains heuristics that are applied to packing problems. They often yield good (although not provably good) solutions, within a small amount of time.

The focus in this thesis is on approximation algorithms. These algorithms provide near-optimal solutions, and there is a proven bound on the solution quality. For example, for the strip packing problem, Baker *et al.* [BCR80] proved that the algorithm that always chooses the bottommost and leftmost position is a 3-approximation. This means that the height of the packing produced by the algorithm is at most three times higher than an optimal

[1] www.din.de

packing. Moreover, they provide an example in which the factor of three is actually achieved. Hence, there is an upper bound and a lower bound of 3 on the solution quality. A survey on approximation algorithms for packing problems can be found in [Ste08].

Outline of this Thesis In Chapter 2 we present basic definitions used throughout this work.

Chapter 3 studies the problem of dynamically inserting and deleting blocks from an array. The blocks can be moved inside the array and our goals are to minimize the time until the last block is removed and the costs for the block moves. This problem differs from other *storage allocation problems* in particular in the way the blocks can be moved. We present complexity results, different algorithms with provably good behavior, and provide computational experiments.

A variant of the strip packing problem with additional constraints—*Tetris constraint* and *gravity constraint*—is studied in Chapter 4. We present two algorithms achieving *asymptotic competitive factors* of 3.5 and 2.6154, respectively. These algorithms improve the best previously known algorithm, which achieves a factor of 4.

In Chapter 5 we present two closely related problems. They both have in common that point sets with small interior distances have to be selected. In particular, the first problem asks for the selection of grid points from the two-dimensional integer grid such that the average pairwise L_1 distances are minimized. We present the first optimal algorithm for this problem. In the second problem, we have to pack shapes with fixed area into a unit square, minimizing, again, the distances inside the shapes. We present a 5.3827-approximation algorithm.

Every chapter starts with a problem statement and definitions needed in the rest of the chapter. Moreover, we present work related to the problem, at the beginning of every chapter. Additionally, Chapter 3 contains related work at the beginning of the Sections 3.3 and 3.4.

Three papers form the basis of this thesis. All of them were prepared in collaboration with other people. Chapter 3 is based on the paper [BFKS09] and Chapter 4 on the paper [FKS09]. Both were prepared together with **Sándor P. Fekete** and **Tom Kamphans**.

The paper [DFR+09] forms the basis for Section 5.2. It was prepared in collaboration with **Erik D. Demaine**, **Sándor P. Fekete**, **Günther Rote**, **Daria Schymura**, and **Mariano Zelke**.

Chapter 2

Preliminaries

In this chapter we review some basic definitions from complexity theory and the analysis of algorithms. We explain the general concepts rather than defining these terms in detail. Good textbooks, with rigorous definitions, are provided by Papadimitriou [Pap94] and by Garey and Johnson [GJ79]. Definitions that are only used in a particular chapter, are stated at the beginning of each chapter.

A *problem* consists of a set of parameters and a question that has to be answered. An *instance* of a problem is a precise specification of the parameters. The *input size* of an instance is the number of bits needed to store the parameters, *e.g.,* in *binary encoding* (using only zeros and ones) a number $n \in \mathbb{N}$ requires $O(\log n)$ bits. We say that an algorithm solves a problem in *polynomial time* if its running time for any instance is bounded by a polynomial in the input size of that instance. All problems admitting a polynomial time algorithm are contained in the class P. An algorithm with running time polynomially bounded in the input size and the maximum input value is called *pseudo-polynomial*.

A problem is a *decision problem*, if the answer to the question is either "yes" or "no". A decision problem belongs to the class NP, if there exists a certificate for every "yes" instance that can be checked in polynomial time, *i.e.,* a given solution can be checked in polynomial time. It is obvious that $\mathrm{P} \subseteq \mathrm{NP}$, but it is not clear whether $\mathrm{P} = \mathrm{NP}$. It is widely believed that $\mathrm{P} \neq \mathrm{NP}$. A decision problem π is called NP-*hard* if every instance of a problem $\pi' \in \mathrm{NP}$ can be transformed into an instance of π in polynomial time. This transformation is usually called *reduction*. A decision problem is called NP-*complete* if it is NP-hard and a member of NP.

All these terms are commonly also used for *optimization problems*, *i.e.,* for problems that ask for a minimum or a maximum solution (*minimization* and *maximization problem*).

Consider a minimization problem and a polynomial time algorithm ALG solving the problem. We call ALG an *asymptotic α-approximation algorithm* (or *asymptotic α-approximation* for short), if

$$|\mathrm{ALG}| \leq \alpha \cdot |\mathrm{OPT}| + \beta \;, \tag{2.1}$$

where $|\mathrm{ALG}|$ is the value computed by ALG, and $|\mathrm{OPT}|$ is the value of an optimal solution; moreover, β must be a constant whereas α may be a constant or depend on the input size. If β is equal to zero, then ALG is called an *absolute α-approximation (algorithm)*; α is called the *approximation factor* or *approximation ratio*. For maximization problems the roles of $|\mathrm{ALG}|$ and $|\mathrm{OPT}|$ are exchanged in Eq. (2.1). A *fully polynomial time approximation scheme* is an algorithm that is a $(1+\varepsilon)$-approximation for any $\varepsilon > 0$, running in time polynomially bounded in the input size of the instance and the input size of ε, as well as in the value $\frac{1}{\varepsilon}$. For further reading we recommend the book by Hochbaum [Hoc97].

The performance of an online algorithm is measured in a similar way; again, we consider a minimization problem. We call an online algorithm ALG an *asymptotic c-competitive* algorithm (or *asymptotic c-competitive* for short), if there exits a constant β such that

$$|\mathrm{ALG}| \leq c \cdot |\mathrm{OPT}| + \beta \;,$$

where $|\mathrm{ALG}|$ is the value computed by ALG and $|\mathrm{OPT}|$ is the value of an optimal solution. If $\beta = 0$, then ALG is called *absolute c-competitive*; c is called the *competitive factor* or the *competitive ratio*, and c can be a constant or depend on the input size. A textbook on online algorithms is provided by Fiat and Woeginger [FW98].

If we speak of algorithms in general (not distinguishing between online and approximation algorithms) we also use the term *performance ratio* to refer to α or c, respectively. The smaller α and c are for an algorithm, the better is its overall behavior compared to the optimum.

There are several standard techniques to design an algorithm, *e.g., divide-and-conquer*, *dynamic programming*, and so on. In Section 5.2 we will use dynamic programming to compute optimal solutions for a packing problem. The main idea of this approach is that optimal solutions for the whole problem can be obtained from solutions to (smaller) subproblems. To avoid that the smaller problems are solved more than once, their solution values are usually stored in a table. For an excellent introduction to dynamic programming see the book by Cormen *et al.* [CLR90].

Chapter 3

Maintaining Arrays of Contiguous Objects

In this chapter we consider methods for dynamically storing a set of one-dimensional objects ("blocks") of different size inside an array. Blocks are inserted and removed, resulting in a fragmented layout that makes it harder to insert further blocks. It is possible to relocate blocks to another free subarray that is contiguous and does not overlap with the current location of the block. These constraints distinguish our problem from the classical memory allocation problem. Our goals are to minimize the time until the last block is removed from the array and the costs for moving the blocks.

3.1 Model and Problem Description

Let A be an array of length $|A|$ and $B_1, \ldots, B_n$ a sequence of blocks which must be inserted into the array. The blocks arrive online, *i.e.,* the next block arrives after the previous one has been inserted. Every block, B_i, is characterized by its *size*, b_i, and its *processing time*, t_i. A *free space* is a subarry of A, of maximal size, not containing a block or a part of a block. A block, B_i, can be inserted into A, occupying a subarray, A_i, of size b_i, if A_i is a subset of a free space.

After the insertion, a block stays t_i time units in the array before it is removed; this duration is not known when placing the block. It becomes known when the block is removed. If a block cannot be placed, *i.e.,* there is no free space of appropriate size, it must wait until the array is compacted or other blocks are removed. We assume that the first block is inserted at time $T = 0$, and we define the *makespan* as the number of time units until the last block is removed from the array.

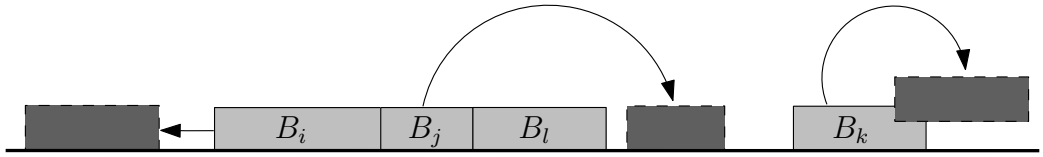

Figure 3.1: The four blocks B_i, B_j, B_k, and B_l each occupying a subarray of A. The blocks B_i and B_j are moved to new positions: B_i is shifted, and B_j is flipped. The move of block B_k is forbidden, because the current and the target position overlap. If these kind of moves would be allowed connecting the total free space could *always* be done by shifting all blocks to one side.

The blocks can be moved inside the array. A block located in a subarray, A_s, can be *moved* to a subarray, A_t, if the following two constraints are satisfied:

(I) A_t is of the same size as A_s, and

(II) A_t is part of a free space.

Note that the second property implies that both subarrays, A_s and A_t, are disjoint. This property makes the model different from the ones studied in classical storage allocation scenarios.

We distinguish moves into *flips* and *shifts*: If there is at least one block located between A_s and A_t, we call the move a flip, otherwise a shift; see Fig. 3.1.

Moves are charged using a function, $c(b_i)$. For example, we can simply count the *number of moves* using $c_1(b_i) := 1$, or we count the *moved mass* with $c_2(b_i) := b_i$. The *total cost* of a sequence of moves is simply the sum of the costs per move. We define the *Sequence Insertion Problem (SIP)* studied in this chapter:

Given an array, A, and a sequence of blocks, $B_1, \ldots, B_n$, arriving online. The task is to insert all blocks into A such that the makespan *and* the total costs of the moves are minimized.

We need some definitions: We call a fixed arrangement of blocks and free spaces inside an array an *allocation*. For a given allocation, we always denote by f the total size of all free spaces and by $f_{\max}$ and $b_{\max}$ the size of the largest free space and the largest block, respectively.

Organization of this Chapter The remainder of the current section shows two applications of the SIP and presents related work. In Section 3.2 we follow the question whether it pays of to delay moves as far as possible, *i.e.,* until the next block cannot be inserted. It turns out that in general it

is NP-hard to decide whether the next insertion is possible. If the insertion is possible, $O(n^2)$ moves might be necessary to create a sufficiently large free space. Therefore, we focus on strategies that organize the array such that these "bad" situations are avoided. We present a certificate that guarantees a minimum makespan in Section 3.3. Finally, in Section 3.4, we present and evaluate different strategies.

As far as we know, no work has been done on the SIP so far. We published the results presented in this chapter in [BFKS09] and [AFK+08].

Applications There are two main applications that initiated our research. The first application is to maintain blocks on a *Field Programmable Gate Array (FPGA)* a reconfigurable chip that consist of a two-dimensional array of processing units; see [Mey07] for a textbook on FPGAs.

Each unit can perform one basic operation depending on its configuration, which can be changed during runtime. A block is a configuration for a set of processing units wired together to fulfill a certain task. As a lot of FPGAs allow only whole columns to be reconfigured, we allow the blocks to occupy only whole columns on the FPGA (and deal with a one-dimensional problem). Moreover, because the layout of the blocks (*i.e.,* configurations and interconnections of the processing units) is fixed, we have to allocate *connected* free space for a block on the FPGA.

In operation, different blocks are loaded onto the FPGA, executed for some time and are removed when their task is fulfilled, causing fragmentation on the FPGA. When fragmentation becomes too high (*i.e.,* we cannot place blocks, although there is sufficient free space, but no sufficient amount of connected free space), the execution of new tasks has to be delayed until other tasks are finished and the corresponding blocks are removed from the FPGA. To reduce the delay, we may reduce fragmentation by moving blocks. Moving a block means to stop its operation, copy the block to an unoccupied space, restart the block in the new place, and declare the formerly occupied space of the block as free space; see Fig. 3.1. Thus, it is important that the current and the target position of the block are *not overlapping* (*i.e.,* they do not share a column).

The second application is closely related to storage allocation but with a strong emphasis on fast and secure data management. For example *Tokutek*[1], a company working in the field of databases and file systems, considers these issues.

[1] `http://tokutek.com`

As in classical storage allocation scenarios, blocks are inserted and removed from a memory causing fragmentation. Modern memory allocation algorithms use pointer to store data in a fragmented array; see [SPG94] for a textbook. However, if the memory (*e.g.,* a hard disk) needs to perform time-consuming rotations to follow a pointer, algorithms that read a lot of data from the memory become rather slow. Therefore, one wants to avoid the use of "too many" pointers. In addition, an incorrectly set pointer or the loss of a pointer due to memory failure causes the loss of parts of blocks stored in the memory. Hence, in a fast and secure data management system pointers should be used only rarely or not at all. If no pointers are used, all blocks have to be stored in *contiguous* subarrays.

To make sure that an error which might occur during the movement of a block does not cause loss of data we further require that before a block is deleted there is a full copy of it somewhere in the memory. In other words, the current position and the target position of a block have to be *disjoint*.

Related Work Of course, the SIP is closely related to the well studied *dynamic storage allocation problem*. The objective (minimizing the makespan) is the same as in the SIP. The two problems differ from each other whether blocks can be moved at all and if so what kind of moves are allowed. In the latter case, minimizing the costs for the moves becomes an additional goal.

In the setting where blocks cannot be moved a large variety of methods and results has been proposed; see [Knu97a] for a textbook and [WJNB95] for a survey. The two most commonly used techniques are sequential fit algorithms and buddy methods. The sequential fit algorithms, *e.g.,* first fit, best fit, and worst fit maintain a list of free spaces and allocate the next block in the first free space, the free space where the block fits best, or worst, respectively. Their worst-case performance is analyzed, *e.g.,* in [LNO96, Rob77, Ger96]; a probabilistic analysis is given, *e.g.,* in [CFL90, CKS85].

Buddy systems partition the storage into a number of standard block sizes and allocate a block in a free interval of the smallest standard size sufficient to contain the block. Differing only in the choice of the standard size, different buddy systems have been proposed [Bro80, Hin75, Hir73, Kno65, SP74].

In the setting where blocks can be moved inside the array constraint (II) appears in a weaker version: A block can be moved to a subarray which might intersect with the subarray currently occupied by the block. If inserting and moving a block, B_i, costs b_i, it is NP-complete to insert a sequence of blocks into the array at minimum cost [BC84]. An algorithm with cost of $b_i(1 + \log b_i/2)$ per insertion (if b_i is a power of 2) is presented in [BCW85].

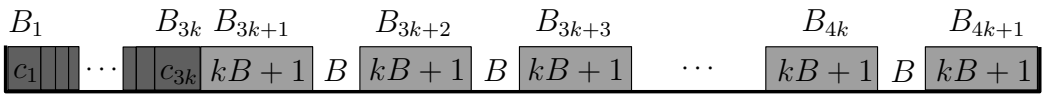

Figure 3.2: We can insert a block of size kB if and only if we can move the first $3k$ blocks (dark-gray) to the k free spaces of size B. The gray blocks cannot be moved.

3.2 Delaying Moves

Delaying moves until no further insertion is possible, could be one way to minimize the total costs for the moves. In this section we analyze the complexity of a single insertion in general and in the situation where the array is only sparsely filled. Furthermore, we proof a lower bound on the number of moves that might be necessary to create a sufficiently large free space for the next insertion.

3.2.1 Complexity of a Single Insertion

We study the problem of inserting a single block into an array that already contains some blocks. More precisely, given a block, B_{n+1}, and an array, A, containing n blocks, $B_1, \ldots, B_n$, we want to answer the question "Does there exist a sequence of moves such that B_{n+1} can be inserted into A?" We call this problem the *Block Insertion Problem (BIP)*. The optimization version of the BIP is to find a sequence of minimum length.

If there is a free space of size at least b_{n+1} or the sum of the sizes of all free spaces is less than b_{n+1} the above question is trivially answered. However, in general it is not that easy:

Theorem 1. *In general, the* BIP *is* NP*-hard.*

Proof. We use a reduction from the 3-PARTITION problem: Given a set of positive integers $c_1, \ldots, c_{3k}$, a bound $B \in \mathbb{N}$ such that c_i satisfies $\frac{B}{4} < c_i < \frac{B}{2}$ for $i = 1, \ldots, 3k$, and $\sum_{i=1}^{3k} c_i = kB$. Can the c_i's be partitioned into k disjoint sets, $S_1, \ldots, S_k$, such that $\sum_{c \in S_j} c = B$, for all $j \in \{1, \ldots, k\}$? This problem is NP-complete [GJ79].

For the reduction, we place $3k$ blocks, $B_1, \ldots, B_{3k}$, with $b_i = c_i$, for $i = 1, \ldots, 3k$, side by side starting at the left end of A. Then, starting at the right boundary of B_{3k}, we place $k+1$ blocks of size $kB+1$, alternating with k free spaces of size B; see Fig. 3.2. We choose $b_{n+1} = kB$. Because the total free space is less than $kB + 1$, only the first $3k$ blocks can ever be moved. Hence, the insertion of B_{n+1} is possible if and only if the first $3k$ blocks are moved to the k free spaces which implies a solution to the 3-PARTITION instance. □

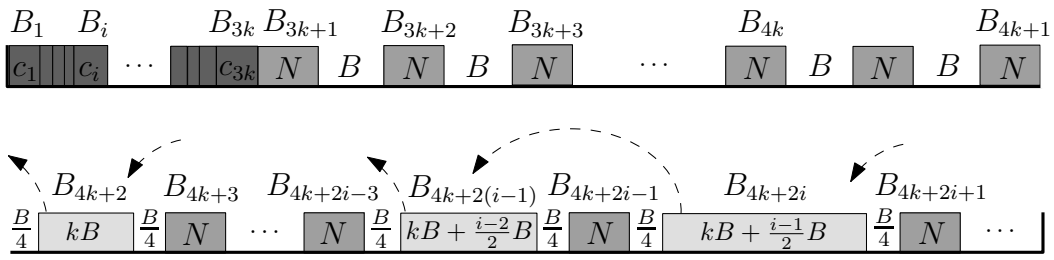

Figure 3.3: The blocks of size $N = kB + 1 + rB/2$ (gray) can never be moved. B_{4k+2} of size kB can be moved if the first $3k$ blocks (dark-gray) can be moved to the first k free spaces of size B. Then, for $i = 2, \ldots, r$, the blocks B_{4k+2i} (light-gray) fit exactly between the blocks $B_{4k+2i-3}$ and $B_{4k+2i-1}$, increasing the size of the maximal free space by $B/2$ with every move.

In the above construction an insertion is possible if the blocks can be rearranged such that there is exactly one free space. The hardness of this problem gives rise to the question whether an insertion is easier if only a fraction of the total free space is required. In other words, "Can the size of the largest constructible free space be approximated?" That is, we want to find a polynomial time algorithm that constructs a free space of size f_{alg} such that $f_{\text{opt}} \leq \alpha \cdot f_{\text{alg}} + \beta$, where f_{opt} denotes the size of the largest free space that can be constructed. Recall, that the input size of the problem is in $O(n' \cdot \max\{\log f_{\max}, \log b_{\max}\}$, where n' is the total number of blocks and free spaces.

Theorem 2. *Let* alg *be a polynomial time algorithm with* $f_{\text{opt}} \leq \alpha \cdot f_{\text{alg}} + \beta$. *Unless* P = NP, $\alpha \in \Omega([n' \cdot \max\{\log f_{\max}, \log b_{\max}\}]^{1-\varepsilon})$, *for any* $\varepsilon > 0$.

Proof. We will show that if alg is an α-approximation algorithm it can be used to decide a 3-PARTITION instance. For a given 3-PARTITION instance with numbers $c_1, \ldots, c_{3k}$ and a bound $B \in \mathbb{N}$ (recall that $\frac{B}{4} < c_i < \frac{B}{2}$) we construct the following allocation of blocks inside an array (see Fig. 3.3): Starting at the left end of the array we place $3k$ blocks side by side with $b_i = c_i$, for $i = 1, \ldots, 3k$. Then, starting at the right boundary of B_{3k}, we place $k + 1$ blocks of size $N = kB + 1 + rB/2$, $r \in \mathbb{N}$, alternating with k free spaces of size B. Now, for $i = 1, \ldots, r$, we proceed with a free space of size $B/4$, a block of size $b_{4k+2i} = kB + (i-1)B/2$, a free space of size $B/4$, and a block of size $b_{4k+2i+1} = N$.

Note that $n' = 5k + 4r + 1$ and $\max\{\log f_{\max}, \log b_{\max}\} = \log b_{\max}$. We claim that $f_{\text{alg}} \geq kB$ if and only if the answer to the 3-PARTITION instance is "yes".

If $f_{\text{alg}} \geq kB$: Consider the situation when there is a free space of size kB for the first time. Because none of the blocks, $B_{3k+1}, \ldots, B_{4k+2r+1}$, could be

moved so far, and because the blocks, $B_1, \ldots, B_{3k}$, are larger than $B/4$ the only way to create a free space of size kB is to place the first $3k$ blocks in the k free spaces of size B; implying a solution to the 3-PARTITION instance.

If $f_{\text{alg}} < kB$: We will show that the answer to the 3-PARTITION instance is "no". If $f_{\text{alg}} < kB$, then

$$f_{\text{opt}} \leq \alpha \cdot f_{\text{alg}} + \beta < kB \cdot C \cdot (n' \log b_{\max})^{1-\varepsilon} + \beta \ ,$$

for some constant C. The total free space has size $f = kB + rB/2$. Because $n' = 5k + 4r + 1$, $b_{\max} = N = kB + 1 + rB/2$ and k, B, C, ε, and β are constant a straightforward computation shows that

$$f_{\text{opt}} < kB \cdot C \cdot \left[(5k + 4r + 1) \log(kB + 1 + \frac{rB}{2})\right]^{1-\varepsilon} + \beta \leq kB + \frac{rB}{2} = f \ ,$$

for large r. Hence, a free space of size $kB + rB/2$ cannot be constructed.

A solution to the 3-PARTITION instance allows the construction of a free space of size $kB + rB/2$ as follows: The first $3k$ blocks are moved to the k free spaces of size B. Now, B_{4k+2} is moved to the free space of size kB and then, one after the other, B_{4k+2i} is moved between the blocks $B_{4k+2i-3}$ and $B_{4k+2i-1}$, for $i = 2, \ldots, r$; see Fig. 3.3.

We conclude that, if $f_{\text{alg}} < kB$, the 3-PARTITION instance has answer "no". □

3.2.2 Moderate Density

The construction in the previous section needs lots of immobile blocks that cannot be moved in the initial allocation, and the size of the total free space is small compared to the size of the array, meaning that the array is densely filled.

We define for an array, A, of length, $|A|$, containing blocks, $B_1, \ldots, B_n$, the *density* as $\delta = \frac{1}{|A|} \sum_{i=1}^{n} b_i$. We show that if at least one of the inequalities

$$\delta \leq \frac{1}{2} - \frac{1}{2|A|} \cdot b_{\max} \text{ or} \tag{3.1}$$

$$b_{\max} \leq f_{\max} \tag{3.2}$$

is satisfied the BIP can be solved with at most $2n$ moves by Algorithm 1. This algorithm simply shifts all blocks to the left and, afterwards, to the right as far as possible. We need two observations to prove its correctness. First, if Eq. (3.1) is satisfied, then

$$f \geq \frac{|A|}{2} + \frac{1}{2} \cdot b_{\max} \ , \tag{3.3}$$

Algorithm 1: LeftRightShift

Input: An array, A, with n blocks, $B_1, \ldots, B_n$, (denoted from left to right) such that Eq. (3.1) or Eq. (3.2) is satisfied.

Output: A placement of $B_1, \ldots, B_n$ such that there is only one free space at the left end of A.

```
for i = 1 to n do
    Shift B_i to the left as far as possible.
end
for i = n to 1 do
    Shift B_i to the right as far as possible.
end
```

because

$$\frac{1}{|A|}(|A| - f) = \frac{1}{|A|}\sum_{i=1}^{n} b_i = \delta \overset{(3.1)}{\leq} \frac{1}{2} - \frac{1}{2|A|} \cdot b_{\max} .$$

Eq. (3.3) follows directly by multiplying both sides with $-|A|$ and then adding $|A|$ on both sides. It follows from Eq. (3.1) that $\delta < \frac{1}{2}$, which implies our second observation:

$$\sum_{i=1}^{n} b_i < f . \tag{3.4}$$

Lemma 1. *Let I be an instance of the* BIP *satisfying Eq. (3.1) or Eq. (3.2). Then, Algorithm 1 creates a single free space of size f.*

Proof. We show that at the end of the first loop the rightmost free space is larger than any block, and therefore, *all* blocks can be shifted to the right in the second loop.

If Eq. (3.1) holds: Let $F_1, \ldots, F_k$ denote the free spaces in A at the end of the first loop. Then, every F_i, $i = 1, \ldots, k-1$, has a block B_j with $b_j > f_i$ to its right because, otherwise, B_j would have been shifted. If this would hold for F_k as well we can conclude that $\sum_{i=1}^{k} f_i < \sum_{i=1}^{n} b_i$ which contradicts (3.4). Hence, there is no block to the right of F_k, and we get

$$\frac{|A|}{2} + \frac{1}{2} \cdot b_{\max} \overset{(3.3)}{\leq} \sum_{i=1}^{k} f_i < \sum_{i=1}^{n} b_i + f_k \overset{(3.1)}{\leq} \frac{|A|}{2} - \frac{1}{2} \cdot b_{\max} + f_k ,$$

implying $b_{\max} < f_k$.

If Eq. (3.2) holds: At least all blocks on the right side of the free space of maximum size are shifted in the first loop. Hence, there is a free space of size at least $b_{\max}$ at the right end of A, afterwards. □

Theorem 3. *Any instance of the* BIP *with* $b_{n+1} \leq f$*, satisfying Eq. (3.1) or Eq. (3.2) can be solved with at most* $2n$ *moves.*

Proof. We use Algorithm 1 to create a free space of size f (Lemma 1). The algorithm moves each of the n blocks at most twice. Afterwards, any block smaller than or equal to f can be inserted. □

The following example shows that the bounds from (3.1) and (3.2) are tight: Consider an array of length 1 containing one block of size $\frac{1}{3}+\varepsilon$ (placed in the middle of the array) and $n-1$ blocks of size ε, for $\varepsilon > 0$, placed at the left end of the array. The large block cannot be moved at all. By construction,

$$\delta = \frac{1}{3} + \varepsilon n \quad \text{and} \quad \frac{1}{2} - \frac{1}{2|A|} \cdot b_{\max} = \frac{1}{3} - \frac{\varepsilon}{2} \, .$$

For fixed n and $\varepsilon \to 0$, these values are arbitrarily close to each other. The same holds for the values

$$f_{\max} = \frac{1}{3} - \frac{\varepsilon}{2} \quad \text{and} \quad b_{\max} = \frac{1}{3} + \varepsilon \, .$$

3.2.3 A Lower Bound on the Number of Moves

So far our aim was to find a sequence of moves that—given an array containing n blocks—rearranges the blocks such that the next block can be inserted; we don't know in advance, if such a sequence exists at all. We saw that in general an appropriate sequence cannot be found in polynomial time, unless P=NP. In this section we assume that there is a sequence that constructs a sufficiently large free space and we want to determine its length.

Theorem 4. *There exists an instance of the* BIP *such that any algorithm needs at least* $\Omega(n^2)$ *moves to solve it.*

Proof. We construct the instance in the following way: For an even number n, we place n blocks, denoted from left to right by $B_1, \ldots, B_n$. The sizes of the blocks are $b_j = b_{n+1-j} = n+2-2j$, for $j = 1, \ldots, n/2$. B_1 has a free space of size $1/2$ to its left, B_n has a free space of size $1/2$ to its right, and every pair of consecutive blocks is separated by a free space of size one, except for the pair $B_{\frac{n}{2}}$ and $B_{\frac{n}{2}+1}$ which is separated by a distance of two; see Fig. 3.4. In this initial allocation, we denote the free spaces from left to right by $F_1, \ldots, F_{n+1}$ and their sizes by $f_1, \ldots, f_{n+1}$. The size of the total free space is $n+1$ and so is the size of the block, B_{n+1}; hence, we need to create a single free space to insert B_{n+1}.

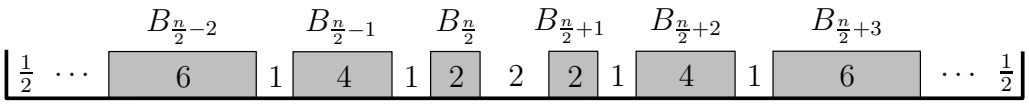

Figure 3.4: The inner part of the instance in the initial allocation. The total free space between two blocks of equal size is equal to the blocks' sizes.

We state two properties of this instance. First, we compute that

$$\sum_{i=j+1}^{n+1-j} f_i = 1 + \sum_{i=j+1}^{n+1-j} 1 = n + 2 - 2j = b_j = b_{n+1-j} \tag{3.5}$$

holds for any pair B_j, B_{n+1-j}, *i.e.,* the total free space between two blocks of equal size is equal to the blocks' sizes; see Fig. 3.4. Second, every block must be moved at least once because of the non-integer free space at the right and the left end of A.

Now, consider a pair of blocks B_{k-1} and $B_{n+1-(k-1)}$, for $k \in \{2, \ldots, \frac{n}{2}+1\}$. We claim that every algorithm needs at least $2n+2k$ moves to make the pair B_{k-1}, $B_{n+1-(k-1)}$ moveable after the pair B_k, B_{n+1-k} can be moved for the first time.

We denote by $A^{(j)}$ the subarray between B_j's right boundary and B_{n+1-j}'s left boundary, for $j = 1, \ldots, n/2$. If B_k and B_{n+1-k} can be moved for the first time, *i.e.,* there is a free space of size at least b_k for the first time, none of the blocks, $B_1, \ldots, B_k, B_{n+1-k}, \ldots, B_n$, has been moved so far. Hence, the largest free space in $A \setminus A^{(k)}$ still has size 1, and therefore, all blocks $B_{k+1}, \ldots, B_{n+1-(k+1)}$ are still placed in $A^{(k)}$, and the free space of size b_k must be in $A^{(k)}$. Because of Eq. (3.5), there is a *single* free space in $A^{(k)}$. Thus, the blocks in $A^{(k)}$ are arranged (described from left to right) as follows: a sequence of blocks lying side by side, a free space of size b_k, a sequence of blocks lying side by side (both sequences might be empty); see Fig. 3.5.

Consider the subarray $A^{(k-1)}$. No block inside $A^{(k-1)}$ can be moved to $A \setminus A^{(k-1)}$ and vice versa. Moreover, no block can be moved inside $A \setminus A^{(k-1)}$. Hence, the only way to make B_{k-1} and $B_{n+1-(k-1)}$ moveable is to create a single free space in $A^{(k-1)}$ by rearranging the blocks inside this subarray. Next we show that every block inside $A^{(k-1)}$ must be moved at least once, to achieve this goal.

Consider a block, B_r, in $A^{(k-1)}$. The distance from B_r's left side to the right side of B_{k-1} is odd because all block sizes and the size b_k of the free space in $A^{(k)}$ are even, and because there is a free space of size 1 between B_k and B_{k-1}; the same holds for the distance from B_r's right side to the left side of $B_{n+1-(k-1)}$. Thus, none of these intervals can be completely filled with other blocks. Because we need to create exactly one free space, this implies

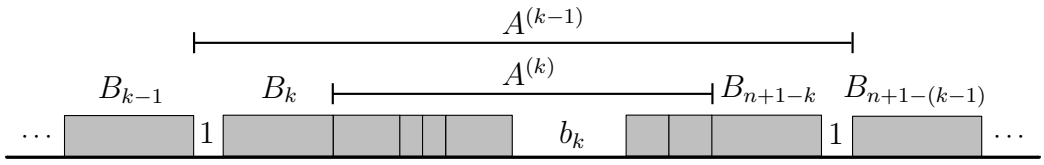

Figure 3.5: The situation when B_k and B_{n+1-k} can be moved for the first time.

that B_{k-1} and $B_{n+1-(k-1)}$ can never become moveable without moving B_r, *i.e.,* without moving all blocks in $A^{(k-1)}$ at least once.

For a pair $B_{j-1}, B_{n+1-(j-1)}$, $j = 2, \ldots, \frac{n}{2}+1$, there are at least $n-2j+2$ blocks in $A^{(j-1)}$. Thus, $\sum_{j=1}^{n/2}(n-2j) = \frac{n^2}{4} - \frac{n}{2}$ is a lower bound on the total number of moves required by any algorithm. □

Of course, there are instances of the BIP where there doesn't exist a sequence of moves such that a sufficiently large free space for the next block can be created. However, given that we know that there is such a sequence, it is an open question whether there is an upper bound that matches our lower bound of $\Omega(n^2)$. It is even open whether there is any polynomial bound.

Open Problem 1. *For any solvable instance of the BIP: "What is the maximum number of moves needed to create a free space such that the next block can be inserted?"*

3.3 Sorting

In this section we study the problem of sorting n blocks according to their size. Sorting is one of the main ingredients to achieve a minimum makespan in the SIP; we prove this in the next section. Moreover, sorting is always an important task that achieved a lot of attention in the literature. Minimizing the makespan is only one of our goals in the SIP; we also want to minimize the costs for the moves. Therefore, we are especially interested in sorting algorithms that use as few moves as possible.

It is necessary to be able to move every block, therefore, we assume in the rest of this section that

$$b_{\max} \leq f_{\max} \tag{3.6}$$

holds, in the initial allocation. If this inequality is not satisfied, there are instances for which it is NP-hard to decide whether the blocks can be sorted or not; this follows from a similar construction as in Section 3.2.1. If thes inequality is fulfilled, Algorithm 1 from the previous section can be used to rearrange the blocks such that there is a single free space at the left end of the array.

Related Work There is a tremendous amount of work on classical sorting, where a set of n elements have to be sorted according to their labels; see, *e.g.,* [Knu97b]. Most of the developed methods rearrange pointers to the elements rather than moving the elements itself. These methods were designed to achieve two goals: minimize the number of comparisons and minimize the auxiliary storage. It is well known that $\Omega(n \log n)$ is a lower bound on the number of comparisons and that at least $\Omega(1)$ auxiliary storage is required by any algorithm. An algorithm using only constant auxiliary storage is called *in-placed* or *in-situ* algorithm. An in-place algorithm using only $O(n \log n)$ comparisons is *Heapsort*.

If the elements have to be sorted physically, meaning that we have to move the elements and not just pointers to them, minimizing the number of moves becomes an additional goal. Obviously, there is a lower bound of $\Omega(n)$ on the number of moves. A fourth goal is to design *stable* sorting algorithms, *i.e.,* elements with equal labels have to be kept in their initial order.

In our setting, we have a free space of size at least $b_{\max}$, and our goal is to minimize the costs for the moves. Of course, minimizing the number of comparisons is necessary to provide a fast implementation, but this is not our main objective. Further, the algorithm does not have to be stable. Hence, we concentrate on in-place sorting algorithms that minimize the number moves:

There is a simple algorithm called *Sorting by Selection* in [Knu97b] and more sophisticated ones in [MR96, FG05]. Speaking in our terms these algorithms assume that n blocks of *unit size* have to be sorted according to their labels. The ideal case is to design an algorithm that is stable, matches the lower bounds for the moves, the comparisons, and the auxiliary storage. This goal hasn't been achieved yet. However, the algorithm from Franceschini and Geffert [FG05] uses $O(n)$ moves, $O(1)$ auxiliary storage, and $O(n \log n)$ comparisons, and there is a stable algorithm from Munro and Raman [MR96] that uses $O(n)$ moves, $O(1)$ auxiliary storage, and $O(n^{1+\varepsilon})$ comparisons for any fixed $\varepsilon > 0$.

The algorithm we describe in this section works similar to *Sorting by Selection* [Knu97b] which works as follows: Assume that there is a single free space at the left end of the array. The algorithm divides the elements into sorted and unsorted ones; at the beginning all elements are unsorted. In every iteration the algorithm moves the unsorted element, E, with the smallest label to the free space. The element that is now to the right of E is moved to E's former position. Sorting by Selection uses $O(n)$ moves, $O(1)$ auxiliary storage and $O(n^2)$ comparisons.

Algorithm 2: Sort

Input: An array, A, containing blocks, $B_1, \ldots, B_n$, and satisfying Eq. (3.6).

Output: The blocks, $B_1, \ldots, B_n$, side by side in sorted order and one free space at the right end of A.

```
1  apply Algorithm 1
2  I := {1, ..., n}
3  while I ≠ ∅ do
4  |  k = argmax_{i∈I}{b_i}, break ties by choosing the leftmost one
5  |  flip B_k to the left end of the free space
6  |  I = I \ {k}
7  |  for i = k − 1, ..., 1 and i ∈ I do
8  |  |  shift B_i to the right as far as possible
9  |  end
10 end
```

3.3.1 Sorting n Blocks with $O(n^2)$ Moves

We will show that Algorithm 2 sorts n blocks with $O(n^2)$ moves if Eq. (3.6) is satisfied. As a first step Algorithm 1 is applied. Afterwards, there is one free space at the left end of A, and all blocks are lying side by side in A. We number the blocks in the resulting position from left to right from 1 to n. The algorithm maintains a list, I, of unsorted blocks. As long as I is not empty, we proceed as follows: We flip the largest unsorted block, B_k, to the left end of the free space, and we shift all unsorted blocks that were placed on the left side of B_k's former position to the right; see Fig. 3.6. Note that at the end of the algorithm there is again only one free space in A.

Theorem 5. *Let A be an array containing n blocks, and let Eq. (3.6) be satisfied. Then, Algorithm 2 sorts the array with $O(n^2)$ moves.*

Proof. The while loop is executed at most n times. In every iteration, there is at most one flip and $n-1$ shifts. This yields an upper bound of n^2 on the total number of moves.

To prove the correctness, we proceed by induction and claim: At the end of an iteration of the while loop, all B_i, $i \notin I$ (*sorted* blocks), lie side by side at the left end of A in decreasing order (from left to right), and all B_i, $i \in I$ (*unsorted* blocks), lie side by side at the right end of A; see Fig 3.6.

This claim is certainly true before the first iteration. Now, consider the situation at the beginning of the j-th iteration of the while loop. Let k be the index of the current maximum in I. By the induction hypothesis and by

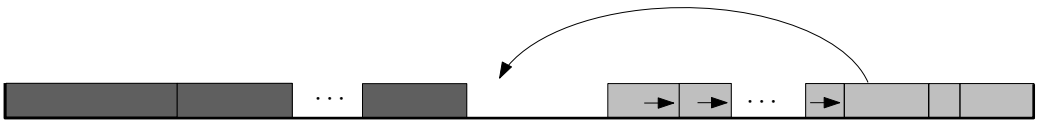

Figure 3.6: One iteration of Algorithm 2. The dark-gray blocks are sorted, and the other ones are unsorted. The largest unsorted block is moved to the left end of the free space, then the unsorted blocks placed at the left side are shifted to the right.

Eq. (3.6), the block B_k can be flipped to the free space. This step increases the number of sorted blocks lying side by side at the left end of A. Since in every step the block of maximum size is chosen, the decreasing order (from left to right) in the sequence of sorted blocks is preserved. Furthermore, this step creates a free space of size b_k that divides the sequence of unsorted blocks into two (possible empty) subsequences. By the numbering of the blocks, the left subsequence contains only indices smaller than k. This ensures that in the second while loop exactly the blocks from the left subsequence are shifted. Again, since B_k is chosen to be of maximum size all shifts are well defined. At the end of the iteration, the unsorted blocks lie side by side and so do the sorted ones. □

3.3.2 A Lower Bound of $\Omega(n^2)$

Now we show that Algorithm 2 needs the minimum number of steps (up to a constant factor) to sort n blocks. In particular, we prove that any algorithm needs $\Omega(n^2)$ steps to sort the following instance: We place an even number of blocks, $B_1, \ldots, B_n$, with size $b_i = k$ if i is odd and $b_i = k + 1$ if i is even, for $k \geq 2$, in A. There is only one free space of size $k+1$ in this initial allocation at the left end of A; see Fig. 3.7. We call blocks of size k *small* and blocks of size $k + 1$ *large*.

Lemma 2. *The following holds for any sequence of shifts and flips applied to the above instance:*

(i) There are never two free spaces, each having a size greater than or equal to k.

(ii) There might be more than one free space but there is always exactly one having either size k or size $k + 1$.

Proof. (i) is obvious, because otherwise, the sum of the sizes of the free spaces would exceed the total free space. (ii) follows because in the last step either a block of size k or $k+1$ was moved, leaving a free of size k or $k+1$, respectively and from (i). □

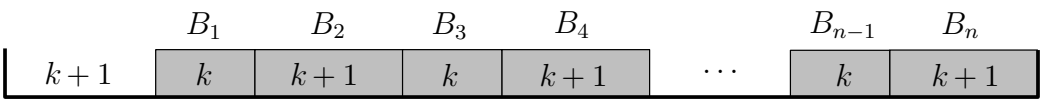

Figure 3.7: Any algorithm needs $\Omega(n^2)$ moves to sort this instance.

Lemma 3. *For any algorithm that uses a minimum number of moves to sort the above instance, the following holds:*

(i) There is never more than one free space in A.

(ii) A small block will only be shifted (and never be flipped).

Proof. (i) Consider a step that creates more than one free space. This is possible only if a block, B_i, of size k was moved, *i.e.,* there is one free space of size k. By Lemma 2, all other free spaces have sizes less than k. Thus, only a block, B_j, of size k can be moved in the next step. Since we only care about the order of the sizes of the blocks not about the order of their labels, the same allocation can be obtained by moving B_j to the current place of B_i and omitting the move of B_i, *i.e.,* the number of steps can be decreased; a contradiction.

(ii) From (i) we know that there is always one free space of size $k+1$. Flipping a small block to this free space creates at least two free spaces. Hence, a small block will only be shifted. □

Theorem 6. *Any algorithm that sorts the above instance needs at least $\Omega(n^2)$ moves.*

Proof. Without loss of generality, we assume that at the end the large blocks are on the left side of the small ones. We consider the array in its initial configuration and, in particular, a small block, B_i, $i \in \{1, 3, \ldots, n-1\}$. There are $\frac{n}{2} - \frac{i-1}{2}$ large blocks and one free space of size $k+1$ to the left of B_i.

Because small blocks are only shifted, the number of small blocks on the left side of B_i will not change but the number of large ones will finally increase to $\frac{n}{2}$. Since a shift moves B_i at most a distance of $k+1$ to the right, B_i must be shifted at least once for every (but one) large block that is moved to B_i's left. Taking the free space into account, this implies that B_i must be shifted at least $\frac{n}{2} - \frac{i-1}{2} - 1$ times, for any odd i between 1 and n. Hence, for $i = 2j-1$ we get a lower bound of $\sum_{j=1}^{n/2}(\frac{n}{2} - j) = \frac{1}{8}n^2 - \frac{1}{4}n$ on the number of shifts. Additionally, every large block must be flipped at least once, because it has a small one to its left in the initial configuration. This gives a lower bound of $\frac{1}{8}n^2 - \frac{1}{4}n + \frac{1}{2}n = \frac{1}{8}n^2 + \frac{1}{4}n$ on the total number of moves for any algorithm. □

Remark We showed that there is a lower bound of $O(n^2)$ on the number of moves to sort blocks according to their size in our model. Note that our algorithm needs auxiliary storage of size $b_{\max}$ and is stable, but requires $O(n^2)$ comparisons.

3.4 Algorithms and Experiments

We studied special aspects of the SIP in the Sections 3.2 and 3.3. The results from Section 3.2 show that there is a strong need to control the process of insertion and deletion to achieve an optimal makespan. In this section we present algorithms for the SIP, give theoretical results for the worst-case performance of the algorithms, and provide experiments.

3.4.1 Related Work

We presented problems closely related to the SIP in Section 3.1. Here, we concentrate on the problem of how to maintain a set of elements in sorted order as this is one of the strategies we use for the SIP.

In the list labeling problem, the task is to maintain a list of at most n items in sorted order. We associate a non-negative number (*label*) less than or equal to $N \in \mathbb{N}$ with each item. We want to maintain monotonic labels in the list, while items are inserted and deleted. An insert operation inserts an item between two items in the list. This can destroy the monotonicity of the labels, and some items need to be relabeled. The cost of an insertion is one plus the number of relabel operations. Depending on N, different upper and lower bounds for the cost of an insertion and a deletion have been proposed:

If N is of size polynomial in n, the algorithms in [Die82, DS87, Tsa84] have amortized cost of $O(\log n)$ per insertion and deletion. A matching lower bound can be found in [DSZ94] and [Zha93].

If N is of size $O(n)$, there is an algorithm with amortized cost $O(\log^2 n)$ per insertion and deletion in [IKR81]. The algorithms in [BCD+02, Wil92] achieve a worst-case cost of $O(\log^2 n)$ per insertion and deletion. For algorithms satisfying a certain smoothness condition Dietz and Zhang prove a matching lower bound [DZ90]; a matching lower bound for all algorithms hasn't been found yet. Bender *et al.* [BFM06] analyze a modified insertion sort and showed that an insertion and a deletion can be performed with cost of $O(\log n)$, with high probability; their work uses ideas from [MG78].

If N is equal to n, there are two algorithms with amortized cost $O(\log^3 n)$ per insertion and deletion in [Zha93, AL90]. Zhang [Zha93] conjectures that there is a matching lower bound.

3.4.2 Algorithms

We turn to the description of our strategies. We present two strategies that achieve an optimal makespan (AlwaysSorted and DelayedSort), a strategy that uses only a constant number of moves and has total cost of $O(b_i \log b_{\max})$ (if $c(b_i)$ is linear in b_i) per insertion and deletion (Classort), as well as a simple heuristic (LocalShift).

AlwaysSorted The main idea of this algorithm is to insert the blocks such that they are sorted according to their size, *i.e.,* the block sizes do not increase from left to right. Note that the sorted order ensures that, if a block, B_i, is removed from the array, all blocks lying on the right side of B_i (these are at most as large as B_i) can be shifted b_i units to the left. In details:

Before a block, B_j, is inserted, we shift all blocks to the left as far as possible, starting at the left end of the array. Next we search for the position that B_j should have in the array to preserve the sorted order. We shift all blocks, lying on the right side of that position, b_j units to the right if possible; after that B_j is inserted.

Theorem 7. *AlwaysSorted achieves the optimal makespan. It performs $O(n)$ moves per insertion in the worst case.*

Proof. All blocks are shifted to the left as far as possible before the next block is inserted. After that, there is only one free space at the right side of A. If this free space is at least as large as the next block, the insertion is performed, meaning that a block must wait if and only if the total free space is smaller than the block size; no algorithm can do better. □

DelayedSort The idea is to reduce the number of moves by delaying the sorting until it is really necessary. We can use Algorithm 2 to sort an array, if $b_{\max} \leq f_{\max}$. Therefore, we try to maintain this property as long as possible, and we switch to sorted order if the condition cannot be preserved:

We maintain a large free space on the left or the right side (alternatingly). First, we check if we can insert the current block B_j at all, *i.e.,* if $b_j \leq f$. Now, if we can insert B_j maintaining $b_{\max} \leq f_{\max}$, we insert B_j using First-Fit; FirstFit starts at the side where we do not keep the free space. Otherwise, we check if B_j can be inserted—maintaining the above condition—after compacting the array by shifting all blocks to the side where we currently keep the large free space, beginning with the block next to the free space. If maintaining the condition is not possible, we sort the array using Algorithm 2 and insert the block into the sorted order (as in AlwaysSorted). Using similar arguments as in the proof of Theorem 7 we get:

Theorem 8. *DelayedSort achieves the optimal makespan. It performs $O(n^2)$ moves per insertion in the worst case.*

ClassSort For this strategy, we assume that the size of the largest block is at most half the size of the array. We round the size of a block, B_i, to the next larger power of 2; we denote the rounded size by b'_i.

We organize the array in $a = \lceil \lg \frac{|A|}{2} \rceil$ *classes*, $C_0, C_1, \ldots, C_a$. Class C_i has *level* i and stores blocks of rounded size 2^i. In addition, each class reserves 0, 1, or 2 (initially 1) *buffers* for further insertions. A buffer of level i is a free space of size 2^i. We store the classes sorted by their level in decreasing order.

The numbers of buffers in the classes provide a sequence, $S = s_a, \ldots, s_0$, with $s_i \in \{0, 1, 2\}$. We consider this sequence as a *redundant binary number;* see Brodal [Bro96]. Redundant binary numbers use a third digit to allow additional freedom in the representation of the counter value. More precisely, the binary number $d_\ell d_{\ell-1} \ldots d_0$ with $d_i \in \{0, 1, 2\}$ represents the value $\sum_{i=0}^{\ell} d_i 2^i$. Thus, for example, 4_{10} can be represented as 100_2, 012_2, or 020_2. A redundant binary number is *regular*, if and only if between two 2's there is one 0, and between two 0's there is one 2. The advantage of regular redundant binary numbers is that we can add or subtract values of 2^k taking care of only $O(1)$ carries, while usual binary numbers with ℓ digits and $11 \ldots 1_2 + 1_2 = 100 \ldots 0_2$ cause ℓ carries.

Inserting and deleting blocks benefits from this advantage: The reorganization of the array on insertions and deletions corresponds to subtracting or adding, respectively, an appropriate value 2^k to the regular redundant binary numbers that represents the sequence S. In details: If a block, B_j, with $b'_j = 2^i$ arrives, we store the block in a buffer of the corresponding class C_i. Initially, the array is empty. Thus, we create the classes, $C_1, \ldots, C_i$, if they do not already exist, reserving one free space of size 2^k for every class C_k. If there is no buffer available in C_i, we have a carry in the counter value; that is, we split one buffer of level $i + 1$ to two buffers of level i; corresponding, for example, to a transition of $\ldots 20 \ldots$ to $\ldots 12 \ldots$ in the counter. Then, we subtract 2^i and get $\ldots 11 \ldots$. Now, the counter may be irregular; thus, we must change another digit. The regularity guarantees that we change only $O(1)$ digits [Bro96]. Similarly, deleting a block with $b'_j = 2^i$ corresponds to adding 2^i to S.

Theorem 9. *ClassSort performs $O(1)$ moves per insertion and deletion in the worst case. Let $c(b_i)$ be a linear function, then the amortized cost for inserting or deleting a block of size b_i is $O(b_i \log b_{\max})$.*

Proof. The number of moves is clear. Now, observe a class, C_i. A block of size 2^i is moved, if the counter of the next smaller class, C_{i-1}, switches from 0 to 2 (for the insertion case). On the one hand, because of the regular structure of the counter, we must insert at least blocks with a total weight of 2^{i-1} before we must move a block of size 2^i again. We charge the cost for this move to theses blocks. On the other hand, we charge every block at most once for every class. As we have $\log b_{\max}$ classes, the stated bound follows. The same argument holds for the case of deletion. Note that we move blocks only, if the free space inside a class is not located on the right side of the class (for insertion) or on the left side (for deletion). Thus, alternatingly inserting and deleting a block of the same size does not result in a large number of moves, because we just imaginarily split and merge free spaces. □

LocalShift Let X and Y be a free space or a block, respectively. We define the distance between X and Y as the number of blocks and free spaces between them. For a free space, F_i, we call the set of blocks or free spaces that are at most at a distance $k \in \mathbb{N}$ from F_i the *k-neighborhood* of F_i. The algorithm LocalShift works as follows: If possible we use BestFit to insert the next block, B_j. Otherwise, we look at the k-neighborhood of any free space (from left to right). If shifting the blocks from the k-neighborhood, lying on the left side of F_i, to the left as far as possible (starting a the left side) and shifting the blocks lying on the right side to the right as far as possible (starting at the right side) would create a free space that is at least as large as B_j, we actually perform these shifts and insert B_j. If no such free space can be created, B_j must wait until at least one block is removed from the array. This algorithm performs at most $2k$ moves per insertion.

3.4.3 Experimental Results

To test our strategies, we generated a number of random input sequences and analyzed the performance of the strategies in an array of size 2^{10}. A sequence consists of 100,000 blocks, each block has a randomly chosen size and processing time. For each sequence, size and processing time are shuffled using different probability distributions. We used the *exponential* and the *Weibull distribution* with different expected values. A probability distribution with distribution function $F(x)$ is *heavy tailed*, iff

$$\lim_{x \mapsto \infty} e^{\lambda x} (1 - F(x)) = \infty, \text{ for all } \lambda > 0 \, .$$

Heavy tailed distributions are used if large events have a significant influence on the modeled system. In [Irl93] it is stated for file sizes on unix systems

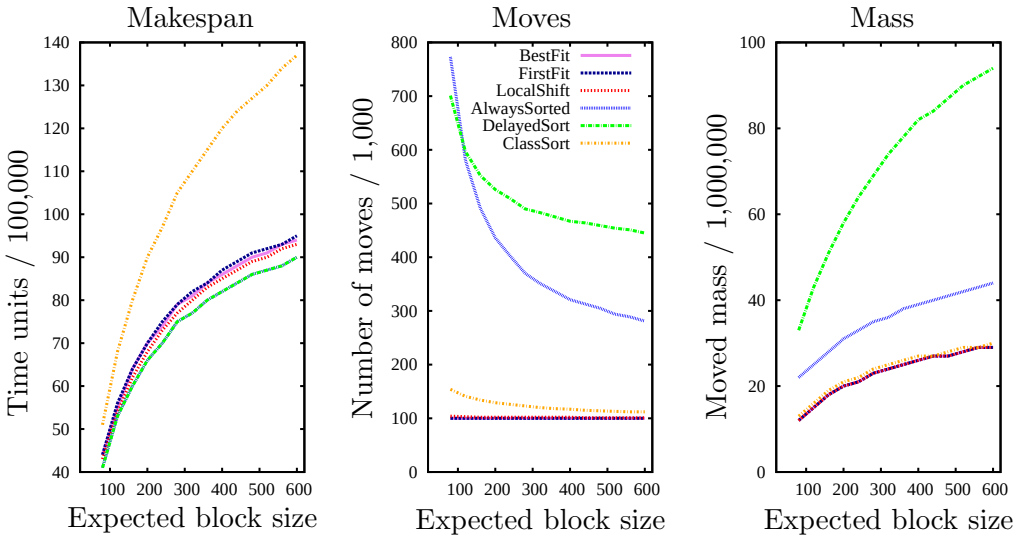

Figure 3.8: The block size is distributed according to the Weibull distribution ($\beta = 0.5$ and $\alpha = 40, \ldots, 300$), and the processing time according to the exponential distribution ($\alpha = 300$).

that 89% of the files take up 11% of the disk space, and 11% of the files take up 89% of the disk space. Moreover, it is stated in [CTB98] that file sizes distributions in the world wide web exhibit heavy tails. Since one of our applications is in the field of memory management we decided to choose the block size according to the Weibull distribution which allows heavy tails. The distribution function of the Weibull distribution is defined as

$$F(x) = 1 - e^{-(\frac{x}{\alpha})^{\beta}} \ .$$

This distribution is heavy tailed, if $\beta < 1$ and it has an expected value of $\alpha \cdot \Gamma(1 + 1/\beta)$, where $\Gamma(z)$ is the gamma function [BN95].

Because the exponential distribution models typical live times [BN95], we chose the processing time according to the exponential distribution. This distribution is a special case of the Weibull distribution with $\beta = 1$; the expected value is α.

We analyzed the two objectives of the SIP: the makespan and the total cost of the moves. We used two different cost functions to count the number of moved blocks ($c(b_i) = 1$) and the moved mass ($c(b_i) = b_i$).

Fig. 3.8 shows the resulting makespan, the number of moves, and the moved mass if the block size is chosen according to the Weibull distribution with $\beta = 0.5$ and α ranging from 40 to 300. This implies that the expected block size ranges from 80 to 600, because $\Gamma(3) = 2$. The processing time is exponentially distributed with parameter $\alpha = 300$.

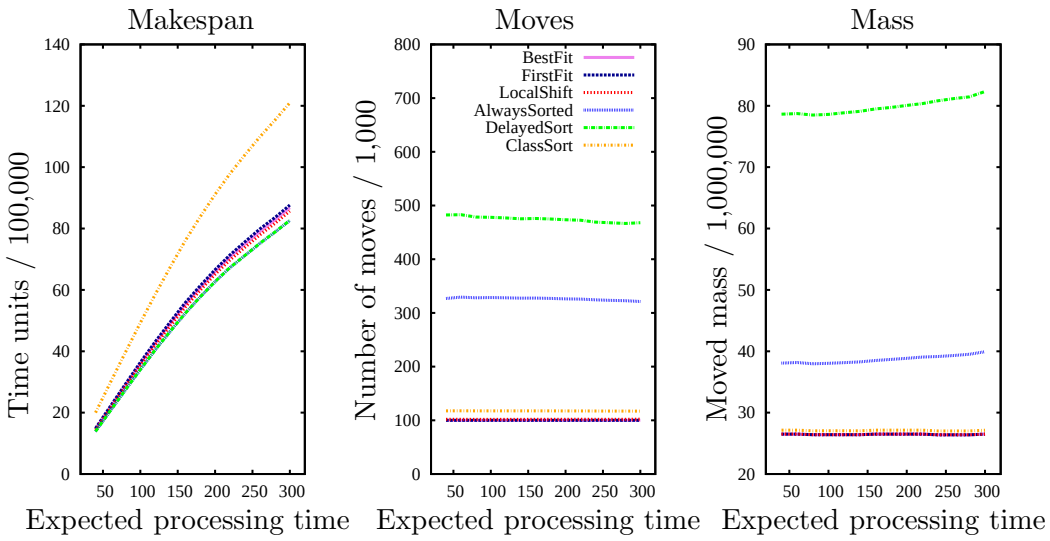

Figure 3.9: The block size is distributed according to the Weibull distribution ($\beta = 0.5$ and $\alpha = 200$), and the processing time according to the exponential distribution ($\alpha = 40, \ldots, 300$).

In Fig. 3.9 the block size is chosen according to the Weibull distribution with $\beta = 0.5$ and $\alpha = 200$. The processing time is exponentially distributed with α ranging from 40 to 300, implying an expected processing time between 40 and 300.

To compare our strategies with strategies that do not move blocks, we also implemented FirstFit and BestFit. Our experiments show that LocalShift—although it is a rather simple strategy—performs very well, as it constitutes a compromise between a moderate number moves and a low makespan. Both, makespan and moves, turn out to be nearly optimal. The size of the neighborhood can be used to balance time and moves. However, increasing the value too much results in a large number of moves, while the makespan decreases only slightly. We tried several sizes of neighborhoods; in our setting (array size 2^{10}), $k = 8$ turned out to be a good choice.

The more complex strategy ClassSort is slightly worse than LocalShift concerning moves, but disappoints in its resulting makespan. In contrast, both types of sorting-related strategies have—of course—a good makespan, but need a lot of moves. Unsurprisingly, First-Fit and Best-Fit need the fewest moves (as they perform moves only on inserting a block, but never move a previously placed block). Their makespan is clearly better than ClassSort, but worse than LocalShift and the sorting strategies.

A comparison of the sorting strategies, AlwaysSorted and DelayedSort, shows that delaying the sorting of the array until it is really necessary doesn't

pay off for the number of moves or the moved mass; this is because switching to sorted order (caused by not enough free space to accompany the largest block) results in a sequence with several moves of the heaviest items. In contrast, AlwaysSorted moves heavy items only rarely.

3.5 Conclusion

We studied the problem of inserting a sequence of contiguous objects in an array. We showed that even a single insertion yields an NP-hard problem. Moreover, the size of the largest constructible free space cannot be approximated within a factor bounded by a sublinear function of the input size. The length of a sequence that constructs a sufficiently large free space can be $\Omega(n^2)$, for an array containing n blocks; we haven't found a matching upper bound yet.

Furthermore, we studied the problem of sorting n blocks inside an array. Our analysis is tight (up to a constant factor): we presented an algorithm that uses $O(n^2)$ moves for any instance and a matching lower bound.

Based on these results, we introduced the strategies AlwaysSorted and DelayedSort which maintain the blocks in the SIP in sorted order. We showed that this is sufficient to achieve an optimal makespan. Thus, we fully achieve one of the goals in the SIP; but, at the cost of a rather large number of moves per insertion ($O(n)$ for AlwaysSorted). The other goal, minimizing the costs for the moves, is well achieved by the algorithms ClassSort and LocalShift; both use only a constant number of moves. Concerning the makespan, LocalShift shows a near-optimal behavior in our experimental evaluation, but ClassSort disappoints because of a large makespan.

Thus, there is still room for improvement. In particular, there might be cheaper certificates than sorting, which allow an optimal makespan and a sublinear number of moves per insertion and deletion.

Chapter 4

Online Square Packing

In this chapter we analyze the problem of packing a sequence of squares into a semi-infinite strip. The squares arrive from above, one at a time, and the objective is to pack them such that the resulting height is minimized. Just like in the classical game of Tetris, each square must be moved along a collision-free path to its final destination. Moreover, we account for gravity in both motion (squares must never move up) and position (any final destination must be supported from below).

4.1 Problem Statement

Let S be a semi-infinite strip of width 1 and $\mathcal{A} = (A_1, \ldots, A_n)$ a sequence of squares with side length $a_i \leq 1$, $i = 1, \ldots, n$. The sequence is unknown in advance. A strategy gets the squares one by one and must place a square before it gets the next. Initially, a square is located above all previously placed ones.

Our goal is to find a non-overlapping packing of squares in the strip that keeps the height of the occupied area as low as possible. More precisely, we want to minimize the distance between the bottom side of S and the highest point that is occupied by a square. The sides of the squares in the packing must be parallel to the sides of the strip. Moreover, a packing must fulfill two additional constraints:

Tetris constraint: At the time a square is placed, there is a collision-free path from the initial position of a square (top of the strip) to the square's final position.

Gravity constraint: A square must be packed on top of another square (*i.e.,* the intersection of the upper square's bottom side and the lower square's top side must be a line segment) or on the bottom of the strip; in addition, no square may ever move up on the path to its final position.

4.1.1 Related Work

In general, a packing problem is defined by a set of items that have to be packed into a container (a set of containers) such that some objective function, *e.g.,* the area where no item is placed or the number of used containers, is minimized. A huge amount of work has been done on different kinds of packing problems. A survey on approximation algorithms for packing problems can be found in [Ste08].

A special kind of packing problem is the *strip packing problem*. It asks for a non-overlapping placement of a set of rectangles in a semi-infinite strip such that the height of the occupied area is minimized. The bottom side of a rectangle has to be parallel to the bottom side of the strip. Over the years, many different variations of the strip packing problem have been proposed: online, offline, with or without rotation, and so on. Typical measures for the evaluation of approximation and online algorithms are the absolute performance and the asymptotic performance ratio.

If we restrict all rectangles to be of the same height, the strip packing problem without rotation is equivalent to the *bin packing problem*: Given a set of one-dimensional items each having a size between zero and one, the task is to pack these items into a minimum number of unit size bins. Hence, all negative results for the bin packing problem, *e.g.,* NP-hardness and lower bounds on the competitive ratio also hold for the strip packing problem; see [GW95] for a survey on (online) bin packing.

If we restrict all rectangles to be of the same width then the strip packing problem without rotation is equivalent to the *list scheduling problem*: Given a set of jobs with different processing times, the task is to schedule these jobs on a set of identical machines such that the makespan is minimized. This problem was first studied by Graham [Gra69]. There are many different kinds of scheduling problems, *e.g.,* the machines can be identical or not, *preemption* might be allowed or not, and there might be other restrictions such as *precedence constraints* or *release times*; see [Bru04] for a textbook on scheduling.

Offline Strip Packing Concerning the absolute approximation factor, Baker *et al.* [BCR80] introduce the class of *bottom-up left-justified* algorithms. A specification that sorts the items in advance is a 3-approximation for a sequence of rectangles and a 2-approximation for a sequence of squares. Sleator [Sle80] presents an algorithm with approximation factor 2.5, Schiermeyer [Sch94] and Steinberg [Ste97] present algorithms that achieve an absolute approximation factor of 2, for a sequence of rectangles.

Concerning the asymptotic approximation factor, the algorithms presented by Coffman *et al.* [CGJT80] achieve performance bounds of 2, 1.7, and 1.5. Baker *et al.* [BBK81] improve this factor to 1.25. Kenyon and Rémila [KR96] design a fully polynomial time approximation scheme. Han *et al.* [HIYZ07] show that every algorithm for the bin packing problem implies an algorithm for the strip packing problem with the same approximation factor. Thus, in the offline case, not only the negative results but also the positive results from bin packing hold for strip packing.

Online Strip Packing Concerning the absolute competitive ratio Baker *et al.* [BS83] present two algorithms with competitive ratio 7.46 and 6.99. If the input sequence consists only of squares the competitive ratio reduces to 5.83 for both algorithms. These algorithms are the first *shelf algorithms:* A shelf algorithm classifies the rectangles according to their height, *i.e.,* a rectangle is in a class s if its height is in the interval $(\alpha^{s-1}, \alpha^s]$, for a parameter $\alpha \in (0, 1)$. Each class is packed in a separate *shelf*, *i.e.,* into a rectangular area of width one and height α^s, inside the strip. A bin packing algorithm is used as a subroutine to pack the items. Ye *et al.* [YHZ09] present an algorithm with absolute competitive factor 6.6623. Lower bounds for the absolute performance ratio are 2 for sequences of rectangles and 1.75 for sequences of squares [BBK82].

Concerning the asymptotic competitive ratio, the algorithms in [BS83] achieve a competitive ratio of 2 and 1.7. Csirik and Woeginger [CW97] show a lower bound of 1.69103 for any shelf algorithm and introduce a shelf algorithm whose competitive ratio comes arbitrarily close to this value. Han *et al.* [HIYZ07] show that for the so called *Super Harmonic* algorithms, for the bin packing problem, the competitive ratio can be transferred to the strip packing problem. The current best algorithm for bin packing is 1.58889-competitive [Sei02]. Thus, there is an algorithm with the same ratio for the strip packing problem. A lower bound, due to van Vliet [Vli92], for the asymptotic competitive ratio, is 1.5401. This bound also holds for sequences consisting only of squares.

Tetris Every reader is certainly familiar with the classical game of Tetris: Given a strip of fixed width, find an online placement for a sequence of objects falling down from above such that space is utilized as good as possible. In comparison to the strip packing problem, there is a slight difference in the objective function as Tetris aims at filling rows. In actual optimization scenarios this is less interesting as it is not critical whether a row is used to precisely 100%—in particular, as full rows do not magically disappear

in real life. In this process, no item can ever move upward, no collisions between objects must occur, an item will come to a stop if and only if it is supported from below, and each placement has to be fixed before the next item arrives. Even when disregarding the difficulty of ever-increasing speed, Tetris is notoriously difficult: Breukelaar *et al.* [BDH+04] show that Tetris is PSPACE-hard, even for the, original, limited set of different objects.

Strip Packing with Tetris Constraint Tetris-like online packing has been considered before. Most notably, Azar and Epstein [AE97] consider online packing of rectangles into a strip; just like in Tetris, they consider the situation with or without rotation of objects. For the case without rotation, they show that no constant competitive ratio is possible, unless there is a fixed-size lower bound of ε on the side length of the objects, in which case there is an upper bound of $O(\log \frac{1}{\varepsilon})$ on the competitive ratio.

For the case in which rotation is possible, they present a 4-competitive strategy based on shelf-packing methods: Each rectangle is rotated such that its narrow side is the bottom side. The algorithm tries to maintain a corridor at the right side of the strip to move the rectangles to their shelves. If a shelf is full or the path to it is blocked, by a large item, a new shelf is opened. Until now, this is also the best deterministic upper bound for squares. Note that in this strategy gravity is not taken into account as items are allowed to be placed at appropriate levels.

Coffman *et al.* [CDW02] consider probabilistic aspects of online rectangle packing without rotation and with Tetris constraint. If n rectangle side lengths are chosen uniformly at random from the interval $[0, 1]$, they show that there is a lower bound of $(0.31382733...)n$ on the expected height for any algorithm. Moreover, they propose an algorithm that achieves an asymptotic expected height of $(0.36976421...)n$.

Strip Packing with Tetris and Gravity Constraint There is one negative result for the setting with Tetris and gravity constraint when rotation is not allowed in [AE97]: If all rectangles have a width of at least $\varepsilon > 0$ or of at most $1 - \varepsilon$, then the competitive factor of any algorithms is $\Omega(\frac{1}{\varepsilon})$.

4.1.2 Our Results

We analyze a natural and simple heuristic called *BottomLeft* (Section 4.2), which works similar to the one introduced by Baker *et al.* [BCR80]. We show that it is possible to give a better competitive ratio than the ratio 4 achieved by Azar and Epstein, even in the presence of gravity. We obtain

an asymptotic competitive ratio of 3.5 for *BottomLeft*. Furthermore, we introduce the strategy *SlotAlgorithm* (Section 4.3), which improves the upper bound to $\frac{34}{13} = 2.6154...$, asymptotically.

We published the results presented in this chapter in [FKS09].

4.2 The Strategy *BottomLeft*

In this section, we analyze the packing generated by the strategy *BottomLeft*, which works as follows: We place the current square as close as possible to the bottom of the strip; this means that we move the square along a collision-free path from the top of the strip to the desired position, without ever moving the square in positive y-direction. We break ties by choosing the leftmost among all possible bottommost positions.

A packing may leave areas of the strip empty. We call a maximal connected component (of finite size) of the strip's empty area a *hole*, denoted by H_h, $h \in \mathbb{N}$. We denote by $|H_h|$ the size of H_h. For a simplified analysis, we finish the packing with an additional square, A_{n+1}, of side length 1. As a result, all holes have a closed boundary. Let $H_1, \ldots, H_s$ be the holes in the packing. We can express the height of the packing produced by BottomLeft as follows:

$$BL = \sum_{i=1}^{n} a_i^2 + \sum_{h=1}^{s} |H_h| \,.$$

In the following sections, we prove that

$$\sum_{h=1}^{s} |H_h| \leq 2.5 \cdot \sum_{i=1}^{n+1} a_i^2 \,.$$

Because any strategy produces at least a height of $\sum_{i=1}^{n} a_i^2$, and because $a_{n+1}^2 = 1$, we get

$$BL = \sum_{i=1}^{n} a_i^2 + \sum_{h=1}^{s} |H_h| \leq \sum_{i=1}^{n} a_i^2 + 2.5 \cdot \sum_{i=1}^{n+1} a_i^2 \leq 3.5 \cdot OPT + 2.5 \,,$$

where OPT denotes the height of an optimal packing. This proves:

Theorem 10. *BottomLeft is (asymptotically)* 3.5*-competitive.*

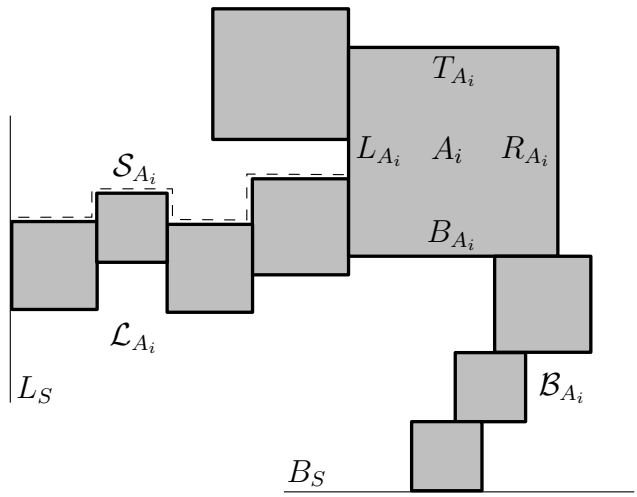

Figure 4.1: The square A_i with its left sequence $\mathcal{L}_{A_i}$, the bottom sequence $\mathcal{B}_{A_i}$, and the skyline $\mathcal{S}_{A_i}$. The left sequence ends at the left side of S, and the bottom sequence at the bottom side of S.

Definitions Before we start with the analysis, we need some definitions: We denote the bottom (left, right) side of the strip by B_S (R_S, L_S; respectively), and the sides of a square, A_i, by B_{A_i}, T_{A_i}, R_{A_i}, L_{A_i} (bottom, top, right, left; respectively); see Fig. 4.1. The x-coordinates of the left and right side of A_i in a packing are l_{A_i} and r_{A_i}; the y-coordinates of the top and bottom side t_{A_i} and b_{A_i}, respectively. Let the *left neighborhood*, $N_L(A_i)$, be the set of squares that touch the left side of A_i. In the same way we define the bottom, top, and right neighborhoods, denoted by $N_B(A_i), N_T(A_i)$, and $N_R(A_i)$, respectively.

A point, P, is called *unsupported*, if there is a vertical line segment pointing from P to the bottom of S whose interior lies completely inside a hole. Otherwise, P is *supported*. A section of a line segment is supported, if every point in this section is supported.

For an object ξ, we refer to the boundary as $\partial\xi$, to the interior as ξ°, and to its area by $|\xi|$. If ξ is a line segment, then $|\xi|$ denotes its length.

Outline of the Analysis We proceed as follows: First, we state some basic properties of the generated packing (Section 4.2.1). In Section 4.2.2 we simplify the shape of the holes by partitioning a hole, produced by Bottom-Left, into several disjoint new holes. In the packing, these new holes are open at their top side, so we introduce *virtual lids* that close these holes. Afterwards, we estimate the area of a hole in terms of the squares that enclose the hole (Section 4.2.3). First, we bound the area of holes that have no virtual

lid and whose boundary does not intersect the boundary of the strip. Then, we analyze holes with a virtual lid; as it turns out, these are "cheaper" than holes with non-virtual lids. Finally, we show that holes that touch the strip's boundary are just a special case. Section 4.2.4 summarizes the costs that are charged to a square.

4.2.1 Basic Properties of the Generated Packing

In this section, we show some basic properties of a packing generated by BottomLeft. In particular, we analyze structural properties of the boundary of a hole.

We say that a square, A_i, *contributes* to the boundary of a hole, H_h, iff ∂A_i and ∂H_h intersect in more than one point, *i.e.,* $|\partial A_i \cap \partial H_h| > 0$. For convenience, we denote the squares on the boundary of a hole by $\tilde{A}_1, \ldots, \tilde{A}_k$ in counterclockwise order starting with the upper left square; see Fig. 4.2. It is always clear from the context which hole defines this sequence of squares. Thus, we chose not to introduce an additional superscript referring to the hole. We define $\tilde{A}_{k+1} = \tilde{A}_1$, $\tilde{A}_{k+2} = \tilde{A}_2$, and so on. By $P_{i,i+1}$ we denote the point where ∂H_h leaves the boundary of $\tilde{A}_i$ and enters the boundary of $\tilde{A}_{i+1}$; see Fig. 4.3.

Let A_i be a square packed by BottomLeft. Then A_i can be moved neither to the left nor down. This implies that either $N_L(A_i) \neq \emptyset$ ($N_B(A_i) \neq \emptyset$) or that L_{A_i} (B_{A_i}) coincides with L_S (B_S). Therefore, the following two sequences $\mathcal{L}_{A_i}$ and $\mathcal{B}_{A_i}$ exist: The first element of $\mathcal{L}_{A_i}$ ($\mathcal{B}_{A_i}$) is A_i. The next element is chosen as an arbitrary left (bottom) neighbor of the previous element. The sequence ends if no such neighbor exits. We call $\mathcal{L}_{A_i}$ the *left sequence* and $\mathcal{B}_{A_i}$ the *bottom sequence* of a square A_i; see Fig. 4.1

We call the polygonal chain from the upper right corner of the first element of $\mathcal{L}_{A_i}$ to the upper left corner of the last element, while traversing the boundary of the sequence in counterclockwise order, the *skyline*, $\mathcal{S}_{A_i}$, of A_i.

Obviously, $\mathcal{S}_{A_i}$ has an endpoint on L_S and $\mathcal{S}^\circ_{A_i} \cap H^\circ_h = \emptyset$. With the help of $\mathcal{L}_{A_i}$ and $\mathcal{B}_{A_i}$ we can prove (see Fig. 4.3):

Lemma 4. *Let $\tilde{A}_i$ be a square that contributes to ∂H_h. Then,*

(i) $\partial H_h \cap \partial \tilde{A}_i$ is a single curve, and

(ii) if ∂H_h is traversed in counterclockwise (clockwise) order, $\partial H_h \cap \partial \tilde{A}_i$ is traversed in clockwise (counterclockwise) order w.r.t. $\partial \tilde{A}_i$.

Proof. For the first part, suppose for a contradiction that $\partial H_h \cap \partial \tilde{A}_i$ consists of at least two curves, c_1 and c_2. Consider a simple curve, C, that lies

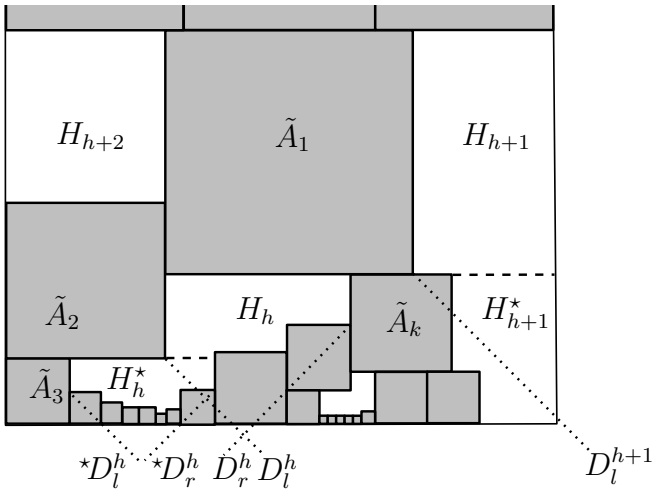

Figure 4.2: A packing produced by BottomLeft. The squares $\tilde{A}_1, \ldots, \tilde{A}_k$ contribute to the boundary of the hole H_h. In the analysis, H_h is split into a number of subholes. In the shown example one new subhole $H^\star_h$ is created. Note that the square $\tilde{A}_1$ also contributes to the holes H_{h+1} and H_{h+2}. Moreover, it serves as a virtual lid for $H^\star_{h+1}$.

completely inside H_h and has one endpoint in c_1 and the other one in c_2. We add the straight line between the endpoints to C and obtain a simple closed curve C'. As c_1 and c_2 are not connected, there is a square, $\tilde{A}_j$, inside C' that is a neighbor of $\tilde{A}_i$. If $\tilde{A}_j$ is a left, right or bottom neighbor of $\tilde{A}_i$ this contradicts the existence of $\mathcal{B}_{\tilde{A}_j}$ and if it is a top neighbor this contradicts the existence of $\mathcal{L}_{\tilde{A}_j}$. Hence, $\partial H_h \cap \partial \tilde{A}_i$ is a single curve.

For the second part, imagine that we walk along ∂H_h in counterclockwise order. Then, the interior of H_h lies on our left-hand side, and all squares that contribute to ∂H_h lie on our right-hand side. Hence, their boundaries are traversed in clockwise order w.r.t. their interior. □

We define P and Q to be the left and right endpoint, respectively, of the line segment $\partial \tilde{A}_1 \cap \partial H_h$. Two squares $\tilde{A}_i$ and $\tilde{A}_{i+1}$ can basically be arranged in four ways, *i.e.,* $\tilde{A}_{i+1}$ can be a left, right, bottom or top neighbor of $\tilde{A}_i$. The next lemma restricts these possibilities:

Lemma 5. *Let $\tilde{A}_i$, $\tilde{A}_{i+1}$ be a pair of squares that contribute to the boundary of a hole H_h.*

(i) If $\tilde{A}_{i+1} \in N_L(\tilde{A}_i)$, then either $\tilde{A}_{i+1} = \tilde{A}_1$ or $\tilde{A}_i = \tilde{A}_1$.

(ii) If $\tilde{A}_{i+1} \in N_T(\tilde{A}_i)$, then $\tilde{A}_{i+1} = \tilde{A}_1$ or $\tilde{A}_{i+2} = \tilde{A}_1$.

Proof. (i) Let $\tilde{A}_{i+1} \in N_L(\tilde{A}_i)$. Consider the endpoints of the vertical line $R_{\tilde{A}_{i+1}} \cap L_{\tilde{A}_i}$; see Fig. 4.3. We traverse ∂H_h in counterclockwise order starting

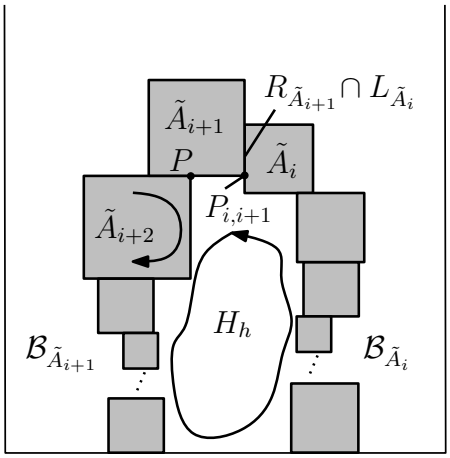

Figure 4.3: The hole H_h with the two squares $\tilde{A}_i$ and $\tilde{A}_{i+1}$ and their bottom sequences. In this situation, $\tilde{A}_{i+1}$ is $\tilde{A}_1$. If ∂H_h is traversed in counterclockwise order then $\partial H_h \cap \partial\tilde{A}_{i+2}$ is traversed in clockwise order w.r.t. to $\partial\tilde{A}_{i+2}$.

in P. By Lemma 4, we traverse $\partial\tilde{A}_i$ in clockwise order, and therefore, $P_{i,i+1}$ is the lower endpoint of $R_{\tilde{A}_{i+1}} \cap L_{\tilde{A}_i}$. Now, $\mathcal{B}_{\tilde{A}_i}$, $\mathcal{B}_{\tilde{A}_{i+1}}$, and the segment of B_S completely enclose an area that completely contains the hole, H_h. If the sequences have a square in common, we consider the area enclosed up to the first intersection. Therefore, if $b_{\tilde{A}_i+1} \geq b_{\tilde{A}_i}$ then $\tilde{A}_{i+1} = \tilde{A}_1$ else $\tilde{A}_i = \tilde{A}_1$ by the definition of $\overline{PQ}$.

The proof of (ii) follows almost directly from the first part. Let $\tilde{A}_{i+1} \in N_T(\tilde{A}_i)$. If ∂H_h is traversed in counterclockwise order, we know that $\partial\tilde{A}_{i+1}$ is traversed in clockwise order, and we know that $\tilde{A}_{i+1}$ must be supported to the left. Therefore, $\tilde{A}_{i+2} \in N_L(\tilde{A}_{i+1}) \cup N_B(\tilde{A}_{i+1})$. Using the first part of the lemma, we conclude that, if $\tilde{A}_{i+2} \in N_L(\tilde{A}_{i+1})$ then $\tilde{A}_{i+2} = \tilde{A}_1$ or $\tilde{A}_{i+1} = \tilde{A}_1$, or if $\tilde{A}_{i+2} \in N_B(\tilde{A}_{i+1})$ then $\tilde{A}_{i+1} = \tilde{A}_1$. □

The last lemma implies that either $\tilde{A}_{i+1} \in N_B(\tilde{A}_i)$ or $\tilde{A}_{i+1} \in N_R(\tilde{A}_i)$ holds for all $i = 2, \ldots, k-2$; see Fig. 4.2. The next lemma shows that there are only two possible arrangements of the squares $\tilde{A}_{k-1}$ and $\tilde{A}_k$:

Lemma 6. *Either* $\tilde{A}_k \in N_R(\tilde{A}_{k-1})$ *or* $\tilde{A}_k \in N_T(\tilde{A}_{k-1})$.

Proof. We traverse ∂H_h from P in clockwise order. From the definition of $\overline{PQ}$ and Lemma 4 we know that $P_{k,1}$ is a point on $L_{\tilde{A}_k}$. If $P_{k-1,k} \in L_{\tilde{A}_k}$, then $\tilde{A}_k \in N_R(\tilde{A}_{k-1})$; if $P_{k-1,k} \in B_{\tilde{A}_k}$, then $\tilde{A}_k \in N_T(\tilde{A}_{k-1})$. In any other case $\tilde{A}_k$ does not have a bottom neighbor. □

Following the distinction described in the lemma, we say that a **hole** is of **Type I** if $\tilde{A}_k \in N_R(\tilde{A}_{k-1})$, and of **Type II** if $\tilde{A}_k \in N_T(\tilde{A}_{k-1})$; see Fig. 4.5.

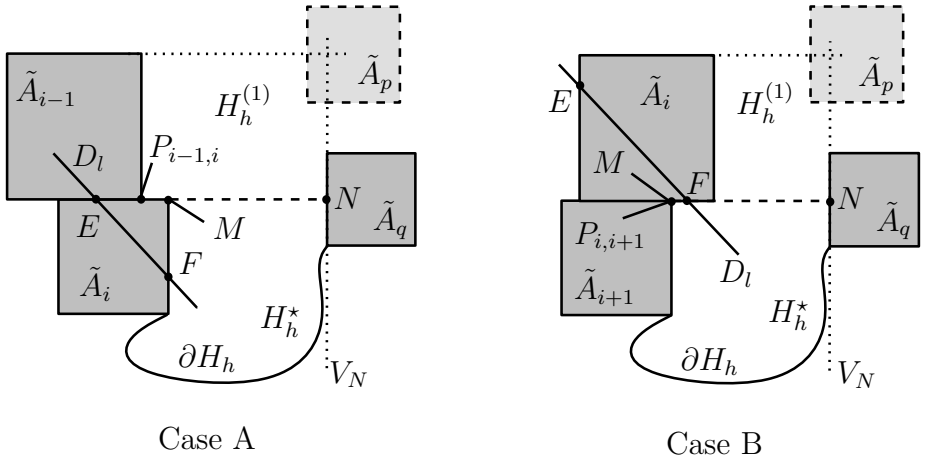

Figure 4.4: D_l can intersect $\tilde{A}_i$ (for the second time) in two different ways: on the right side or on the bottom side. In Case A, the square $\tilde{A}_{i-1}$ is on top of $\tilde{A}_i$; in Case B, $\tilde{A}_i$ is on top of $\tilde{A}_{i+1}$.

4.2.2 Splitting Holes

Let H_h be a hole whose boundary does not touch the boundary of the strip, *i.e.,* the hole is completely enclosed by squares. We define two lines that are essential for the computation of an upper bound for the area of a hole, H_h: The *left diagonal*, D_l^h, is defined as the straight line with slope -1 starting in $P_{2,3}$ if $P_{2,3} \in R_{\tilde{A}_2}$ or, otherwise, in the lower right corner of $\tilde{A}_2$; see Fig. 4.5. We denote the point where D_l^h starts by P'. The *right diagonal*, D_r^h, is defined as the line with slope 1 starting in $P_{k-1,k}$ if $\tilde{A}_k \in N_R(\tilde{A}_{k-1})$ (Type I) or in $P_{k-2,k-1}$, otherwise (Type II). Note that $P_{k-2,k-1}$ lies on $L_{\tilde{A}_{k-1}}$ because otherwise, there would not be a left neighbor of $\tilde{A}_{k-1}$. We denote the point where D_r^h starts by Q'. If h is clear or does not matter we omit the superscript.

Lemma 7. *Let H_h be a hole and D_r its right diagonal. Then, $D_r \cap H_h^\circ = \emptyset$.*

Proof. Consider the left sequence, $\mathcal{L}_{\tilde{A}_k} = (\tilde{A}_k = \alpha_1, \alpha_2, \ldots)$ or $\mathcal{L}_{\tilde{A}_{k-1}} = (\tilde{A}_{k-1} = \alpha_1, \alpha_2, \ldots)$, for H_h being of Type I or II, respectively. It is easy to show by induction that the upper left corners of the α_i's lie above D_r: If D_r intersects $\partial\alpha_i$ at all, the first intersection is on R_{α_i}, the second on B_{α_i}. Thus, at least the skyline separates D_r and H_h. □

If Lemma 7 would also hold for D_l, we could use the polygon formed by D_l, D_r, and the part of the boundary of H_h between Q' and P' to bound the area of H_h, but—unfortunately—it does not.

Let F be the first nontrivial intersection point of ∂H_h and D_l, while traversing ∂H_h in counterclockwise order, starting in P. F is on the boundary of a square, $\tilde{A}_i$. Let E be the other intersection point of D_l and $\partial\tilde{A}_i$.

It is a simple observation that if D_l intersects a square, $\tilde{A}_i$, in a nontrivial way, *i.e.,* in two different points, E and F, then either $F \in R_{\tilde{A}_i}$ and $E \in T_{\tilde{A}_i}$ or $F \in B_{\tilde{A}_i}$ and $E \in L_{\tilde{A}_i}$. To break ties, we define that an intersection in the lower right corner of $\tilde{A}_i$ belongs to $B_{\tilde{A}_i}$. Now, we split our hole, H_h, into two new holes, $H_h^{(1)}$ and $H_h^\star$. We consider two cases (see Fig. 4.4):

- Case A: $F \in R_{\tilde{A}_i} \setminus B_{\tilde{A}_i}$
- Case B: $F \in B_{\tilde{A}_i}$

In Case A, we define $\tilde{A}_{\text{up}} := \tilde{A}_{i-1}$ and $\tilde{A}_{\text{low}} := \tilde{A}_i$, in Case B $\tilde{A}_{\text{up}} := \tilde{A}_i$ and $\tilde{A}_{\text{low}} := \tilde{A}_{i+1}$. Observe the horizontal ray that emanates from the upper right corner of $\tilde{A}_{\text{low}}$ to the right: This ray is subdivided into supported and unsupported sections. Let $U = \overline{MN}$ be the leftmost unsupported section with left endpoint M and right endpoint N; see Fig. 4.4. Now, we split H_h into two parts, $H_h^\star$ below $\overline{MN}$ and $H_h^{(1)} := H_h \setminus H_h^\star$.

We split $H_h^{(1)}$ into $H_h^{(2)}$ and $H_h^{\star\star}$ etc., until there is no further intersection between the boundary of $H_h^{(z)}$ and D_l^h. Because there is a finite number of intersections, this process will eventually terminate. In the following, we show that $H_h^{(1)}$ and $H_h^\star$ are indeed two separate holes, and that $H_h^\star$ has the same properties as an original one, *i.e.,* it is a hole of Type I or II. Thus, we can analyze $H_h^\star$ using the same technique, *i.e.,* we may split $H_h^\star$ w.r.t. *its* left diagonal. We need some lemmas for this proof:

Lemma 8. *Using the above notation we have* $\tilde{A}_{\text{low}} \in N_B(\tilde{A}_{\text{up}})$.

Proof. We consider Case A and Case B separately, starting with Case A. We traverse ∂H_h from F in clockwise order. By Lemma 4, $\tilde{A}_{i-1}$ is the next square that we reach; see Fig. 4.4. Because F is the first intersection, $P_{i-1,i}$ lies between F and E. Thus, either $P_{i-1,i} \in T_{\tilde{A}_i}$ or $P_{i-1,i} \in R_{\tilde{A}_i}$ holds. With Lemma 5, the latter implies either $\tilde{A}_{i-1} = \tilde{A}_1$ or $\tilde{A}_i = \tilde{A}_1$. Because $b_{\tilde{A}_{i-1}} > b_{\tilde{A}_i}$ holds, only $\tilde{A}_{i-1} = \tilde{A}_1$ is possible, and therefore, $\tilde{A}_i = \tilde{A}_2$. D_l intersects $\tilde{A}_2$ in the lower left corner—which is not included in this case—or in $P_{2,3}$. However, $P_{2,3} \in R_{\tilde{A}_2}$ cannot be an intersection, because this would imply $\tilde{A}_3 \in N_R(\tilde{A}_2)$. Thus, only $P_{i-1,i} \in T_{\tilde{A}_i}$ is possible.

In Case B, we traverse ∂H_h from F in counterclockwise order, and $\tilde{A}_{i+1}$ is the next square that we reach. Because F is the first intersection, it follows that $P_{i,i+1}$ lies on $\partial\tilde{A}_i$ between F and E in clockwise order; see Fig. 4.4.

Thus, $\tilde{A}_{i+1} \in N_B(\tilde{A}_i)$ or $\tilde{A}_{i+1} \in N_L(\tilde{A}_i)$ holds. If $\tilde{A}_{i+1} \in N_L(\tilde{A}_i)$, we have $P_{i,i+1} \in L_{\tilde{A}_i}$. If we move from $P_{i,i+1}$ to F on $\partial\tilde{A}_i$, we move in clockwise order on ∂H_h. If we reach $P_{i-1,i}$ before F, the square, $\tilde{A}_{i-1}$, is between $P_{i,i+1}$ and F. The points, $P_{i,i+1}$ and F, are on ∂H_h, and thus, $\partial H_h \cap \tilde{A}_i$ is disconnected, which contradicts Lemma 4. Thus, we reach F before $P_{i-1,i}$. Moreover, $\tilde{A}_i$ must have a bottom neighbor, and therefore, $P_{i-1,i} \in B^\circ_{\tilde{A}_i}$. By Lemma 5, we have $\tilde{A}_i = \tilde{A}_1$ or $\tilde{A}_{i+1} = \tilde{A}_1$. Both cases contradict the fact that D_l intersects neither $\tilde{A}_2$ in the lower right corner nor $\tilde{A}_1$. Altogether, $P_{i,i+1}$ must be on $B_{\tilde{A}_i}$ to the left of F. □

The last lemma states that in both cases, there are two squares for which one is indeed placed on top of the other.

Lemma 9. *M is the upper right corner of $\tilde{A}_{\text{low}}$.*

Proof. Case A: We know $F \in R_{\tilde{A}_i}$ and $P_{i-1,i} \in T_{\tilde{A}_i}$. By Lemma 4, the upper right corner, M', of $\tilde{A}_i$ belongs to ∂H_h. Because F does not coincide with M' (degenerate intersection), $\overline{FM'}$ is a vertical line of positive length. Hence, M' is the beginning of an unsupported section of the horizontal ray emanating from M' to the right. Thus, the first unsupported section starts in M'; that is, $M = M'$. A similar argument holds in Case B. □

To ensure that $H_h^\star$ is well defined, we show that it has a closed boundary. Obviously, $\overline{MN}$ and the part of ∂H_h counterclockwise from M to N forms a closed curve. We place an imaginary copy of $\tilde{A}_{\text{up}}$ on $\overline{MN}$, such that the lower right corner is placed in N. We call the copy the *virtual lid*, denoted by $\tilde{A}'_{\text{up}}$. We show that $\overline{MN} < \tilde{a}_{\text{up}}$ holds, where $\tilde{a}_{\text{up}}$ denotes the side length of $\tilde{A}_{\text{up}}$. Thus, $\overline{MN}$ is completely covered by the virtual copy of $\tilde{A}_{\text{up}}$, and in turn, we can choose the virtual block as a new lid for $H_h^\star$.

Lemma 10. *With the above notation we have $\overline{MN} < \tilde{a}_{\text{up}}$.*

Proof. We show that at the time $\tilde{A}_{\text{up}}$ is packed by BottomLeft, it can be moved to the right along $\overline{MN}$, such that the lower right corner coincides with N. Since $\overline{MN}$ is unsupported, $\overline{MN} \geq \tilde{a}_{\text{up}}$ implies that there would have been a position for $\tilde{A}_{\text{up}}$ that is closer to the bottom of S than its current position.

Let V_N be the vertical line passing through the point N, and let v_N be its x-coordinate. Assume that there is a square, $\tilde{A}_p$, that prevents $\tilde{A}_{\text{up}}$ from being moved. Then, $\tilde{A}_p$ fulfills $l_{\tilde{A}_p} < v_N$ and $b_{\tilde{A}_p} < t_{\tilde{A}_{\text{up}}}$ $(*)$; see Fig. 4.4. Now, consider the sequence $\mathcal{L}_{\tilde{A}_p}$, and note that all squares in $\mathcal{L}_{\tilde{A}_p}$ are placed before $\tilde{A}_{\text{up}}$. From $(*)$ we conclude that the skyline, $\mathcal{S}_{\tilde{A}_p}$, may intersect the

horizontal line passing through $T_{\tilde{A}_{\text{low}}}$ only to the left of v_N. If the skyline intersects or touches in $\overline{MN}$, we have a contradiction to the choice of M and N as endpoints of the first unsupported section. An intersection between M and $P_{\text{up,low}}$ is not possible, because this part completely belongs to $T_{\tilde{A}_{\text{low}}}$. Therefore, $\mathcal{S}_{\tilde{A}_p}$ either intersects the horizontal line to the left of $P_{\text{up,low}}$ or it reaches L_S before. This implies that $\tilde{A}_{\text{up}}$ must pass $\tilde{A}_p$ on the right side and at the bottom side to get to its final position. In particular, $b_{\tilde{A}_p} < t_{\tilde{A}_{\text{up}}}$ implies that $\tilde{A}_{\text{up}}$'s path must go upwards to reach its final position; such a path contradicts the choice of BottomLeft. □

Using the preceding lemmas, we can prove the following:

Corollary 1. *Let $H_h^\star$ and $\tilde{A}'_{\text{up}}$ be defined as above. $H_h^\star$ is a hole of Type I or Type II with virtual lid $\tilde{A}'_{\text{up}}$.*

Proof. $H_h^\star$ has a closed boundary, and there is at least a small area below $\overline{MN}$ in which no squares are placed. Hence, $H_h^\star$ is a hole. Using the arguments that the interior of $\overline{MN}$ is unsupported and that N is supported and lies on $L_{\tilde{A}_q}$, for some $1 \leq q \leq k$, we conclude that there is a vertical line of positive length below N on $\partial\tilde{A}_q$ that belongs to ∂H_h. If we move from N on $\partial\tilde{A}_q$ in counterclockwise order, we move on ∂H_h in clockwise order and reach $\tilde{A}_{q-1}$ next. If $P_{q-1,q} \in L_{\tilde{A}_q}$, then $H_h^\star$ is of Type I. If $P_{q-1,q} \in B_{\tilde{A}_q}$, then it is of Type II. $P_{q-1,q} \notin L_{\tilde{A}_q} \cup B_{\tilde{A}_q}$ yields a contradiction, because in this case there is no bottom neighbor for $\tilde{A}_q$. $\tilde{A}'_{\text{up}}$ is the unique lid by the existence of the sequences $\mathcal{B}_{\tilde{A}_q}$ and $\mathcal{B}_{\tilde{A}_{\text{low}}}$. □

Note that the preceding lemmas also hold for the holes $H_h^{(\dots)}$, $H_h^{\star\star}$, $H_h^{\star\star\star}$, and so on.

Lemma 11. *For every square, A_i, there is at most one copy of A_i.*

Proof. A square, A_i, is used as a virtual lid, only if its lower right corner is on the boundary of the hole that is split. Because its corner can be on the boundary of at most one hole, there is only one hole with virtual lid A_i. □

4.2.3 Computing the Area of a Hole

In this section we show how to compute the area of a hole. In the preceding section we eliminated all intersections of D_l^h with the boundary of the hole, $H_h^{(z)}$, by splitting the hole. Thus, we assume that we have a set of holes, $\hat{H}_h$, $h = 1, \dots, s^\star$, that fulfill $\partial\hat{H}_h \cap D_l^h = \emptyset$ and have either a non-virtual or a virtual lid.

Our aim is to bound $|\hat{H}_h|$ by the areas of the squares that contribute to $\partial\hat{H}_h$. A square, A_i, may contribute to more than one hole. Therefore, it is to expensive to use its total area, a_i^2, in the bound for a single hole. Instead, we charge only fractions of a_i^2 per hole. Moreover, we charge every edge of A_i separately. By Lemma 4, $\partial\hat{H}_h \cap \partial A_i$ is connected. In particular, every side of A_i contributes at most one (connected) line segment to $\partial\hat{H}_h$. For the left (bottom, right) side of a square, A_i, we denote the length of the line segment contributed to $\partial\hat{H}_h$ by λ_i^h (β_i^h, ρ_i^h; respectively). If a side of a square does not contribute to a hole, the corresponding length of the line segment is defined to be zero.

Let $c_{h,i}^{\{\lambda,\beta,\rho\}}$ be appropriate coefficients, such that the area of a hole can be charged against the area of the adjacent squares, *i.e.,*

$$|\hat{H}_h| \leq \sum_{i=1}^{n+1} c_{h,i}^{\lambda}(\lambda_i^h)^2 + c_{h,i}^{\beta}(\beta_i^h)^2 + c_{h,i}^{\rho}(\rho_i^h)^2 \,.$$

As each point on ∂A_i is—obviously—on the boundary of at most one hole, the line segments are pairwise disjoint. Thus, for the left side of A_i, the two squares inside A_i induced by the line segments, λ_i^h and λ_i^g, of two different holes, $\hat{H}_h$ and $\hat{H}_g$, do not overlap. Therefore, we obtain

$$\sum_{h=1}^{s^\star} c_{h,i}^{\lambda} \cdot (\lambda_i^h)^2 \leq c_i^{\lambda} \cdot a_i^2 \,,$$

where c_i^{λ} is the maximum of the $c_{h,i}^{\lambda}$'s taken over all holes $\hat{H}_h$. We call c_i^{λ} the *charge of* L_{A_i} and define c_i^{β} and c_i^{ρ} analogously.

We use virtual copies of some squares as lids. However, for every square, A_i, there is at most one copy, A_i'. We denote the line segments and charges corresponding to A_i' by $\lambda_{i'}^h$, $c_{h,i'}^{\lambda}$, and so on. Taking the charges to the copy into account, the *total charge of* A_i is given by

$$c_i = c_i^{\lambda} + c_i^{\beta} + c_i^{\rho} + c_{i'}^{\lambda} + c_{i'}^{\beta} + c_{i'}^{\rho} \,.$$

Altogether, we bound the total area of the holes by

$$\sum_{h=1}^{s^\star} |\hat{H}_h| \leq \sum_{i=1}^{n+1} c_i \cdot a_i^2 \leq \sum_{i=1}^{n+1} c \cdot a_i^2 \,,$$

where $c = \max_{i=1,\ldots,n}\{c_i\}$. In the following, we want to find an upper bound on c.

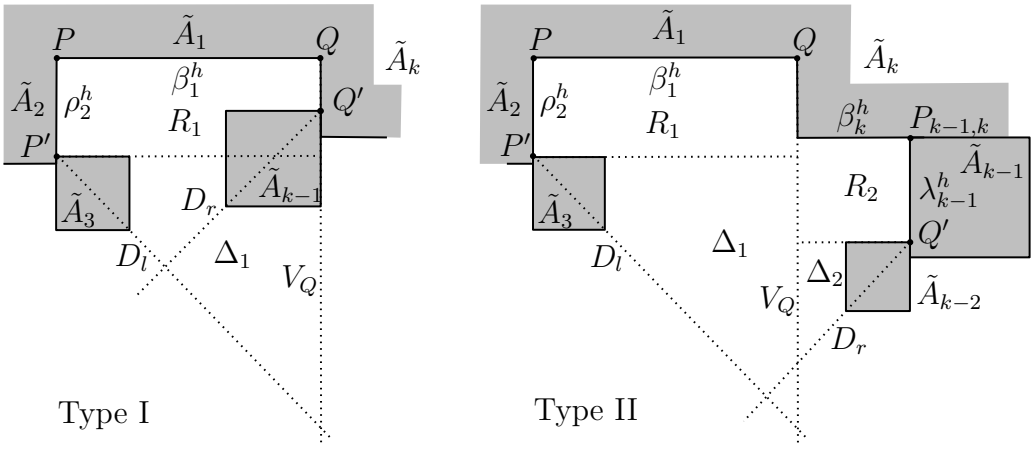

Figure 4.5: Holes of Type I and II with their left and right diagonals.

Holes with a Non-Virtual Lid We know that each hole is either of Type I or II. Moreover, we removed all intersections of $\hat{H}_h$ with its diagonal, D_l^h. Therefore, $\hat{H}_h$ lies completely inside the polygon formed by D_l^h, D_r^h, and the part of $\partial\hat{H}_h$ that is clockwise between P' and Q'; see Fig. 4.5.

If $\hat{H}_h$ is of Type I, then we consider the rectangle, R_1, of area $\rho_2^h \cdot \beta_1^h$ induced by the points P, P', and Q. Moreover, let Δ_1 be the triangle below R_1 formed by the bottom side of R_1, D_l^h, and the vertical line, V_Q, passing through Q; see Fig. 4.5. We obtain:

Lemma 12. *Let $\hat{H}_h$ be a hole of Type I. Then,*

$$|\hat{H}_h| \leq (\beta_1^h)^2 + \frac{1}{2}(\rho_2^h)^2 \,.$$

Proof. Obviously, $|\hat{H}_h| \leq |R_1| + |\Delta_1|$. As D_l^h has slope -1, we get $|\Delta_1| = \frac{1}{2}(\beta_1^h)^2$. Moreover, we have $|R_1| = \rho_2^h \cdot \beta_1^h \leq \frac{1}{2}(\rho_2^h)^2 + \frac{1}{2}(\beta_1^h)^2$. Altogether, we get the stated bound. □

Thus, we charge the bottom side[1] of $\tilde{A}_1$ with 1 and the right side of $\tilde{A}_2$ with $\frac{1}{2}$. In this case, we get $c_{h,1}^{\beta} = 1$ and $c_{h,2}^{\rho} = \frac{1}{2}$.

If $\hat{H}_h$ is of Type II, we define R_1 and Δ_1 in the same way. In addition, R_2 is the rectangle of area $\beta_k^h \cdot \lambda_{k-1}^h$ induced by the points Q' and $P_{k-1,k}$ as well as the part of $B_{\tilde{A}_k}$ that belongs to $\partial\hat{H}_h$. Let Δ_2 be the triangle below

[1]The charge to the bottom of $\tilde{A}_1$ can be reduced to $\frac{3}{4}$ by considering the larger one of the rectangles, R_1 and the one induced by Q, Q', and P, as well as the triangle below the larger rectangle formed by D_l^h and D_r^h. However, this does not lead to a better competitive ratio, because these costs are already dominated by the cost for holes of Type II.

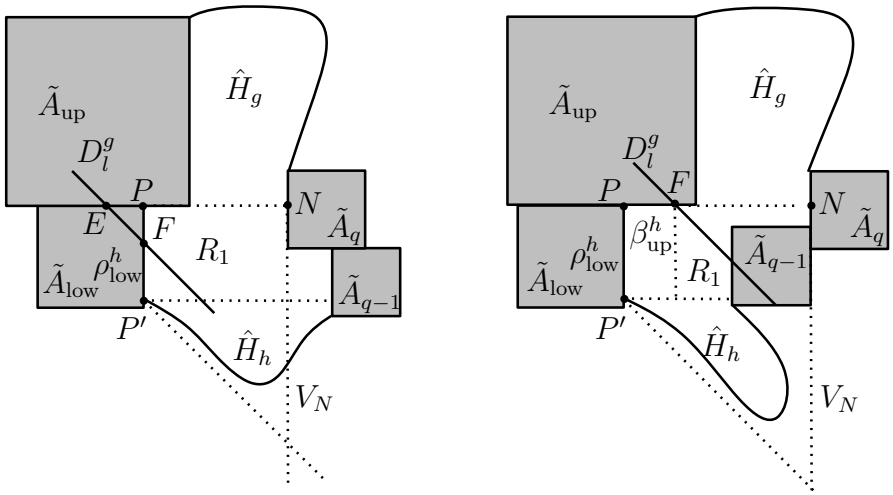

Figure 4.6: The holes $\hat{H}_g$ and $\hat{H}_h$ and the rectangle R_1 which is divided into two parts by D_l^g. The upper part is already included in the bound for $\hat{H}_g$. The lower part is charged completely to $R_{\tilde{A}_{\text{low}}}$ and $B_{\tilde{A}'_{\text{up}}}$. Here P and P' are defined w.r.t. $\hat{H}_h$.

R_2, induced by the bottom side of R_2, D_r^h, and V_Q. Using similar arguments as in the preceding lemma, we get:

Corollary 2. *Let $\hat{H}_h$ be a hole of Type II. Then,*

$$|\hat{H}_h| \leq (\beta_1^h)^2 + (\beta_k^h)^2 + \frac{1}{2}(\rho_2^h)^2 + \frac{1}{2}(\lambda_{k-1}^h)^2 \ .$$

We obtain the charges $c^{\beta}_{h,1} = 1$, $c^{\rho}_{h,2} = \frac{1}{2}$, $c^{\beta}_{h,k} = 1$ and $c^{\lambda}_{h,k-1} = \frac{1}{2}$. Thus, we have a maximum total charge of 2 (bottom: 1, left: 1/2, and right: 1/2) for a square, so far.

Holes with a Virtual Lid Next we consider a hole, $\hat{H}_h$, with a virtual lid. Let $\hat{H}_g$ be the hole immediately above $\hat{H}_h$, *i.e.,* $\hat{H}_h$ was created by removing the diagonal-boundary intersections in $\hat{H}_g$. Corresponding to Lemma 8, let $\tilde{A}_{\text{up}}$ be the square whose copy becomes a new lid, while $\tilde{A}'_{\text{up}}$ is the copy. The bottom neighbor of $\tilde{A}_{\text{up}}$ is denoted by $\tilde{A}_{\text{low}}$. We show that $\tilde{A}'_{\text{up}}$ increases the total charge of $\tilde{A}_{\text{up}}$ not above 2.5. Recall that $\hat{H}_h$ is a hole of Type I or II by Corollary 1.

If $\tilde{A}_{\text{up}}$ does not exceed $\tilde{A}_{\text{low}}$ to the left, it cannot serve as a lid for any other hole; see Fig. 4.6. Hence, the charge of the bottom side of $\tilde{A}_{\text{up}}$ is zero; by Corollary 1, Lemma 12, and Corollary 2 we obtain a charge of at most 1 to the bottom side of $\tilde{A}'_{\text{up}}$. Thus, we get a total charge of 1 to $\tilde{A}_{\text{up}}$. For an

easier summation of the charges at the end, we transfer the charge from the bottom side of $\tilde{A}'_{\text{up}}$ to the bottom side of $\tilde{A}_{\text{up}}$.

If it exceeds $\tilde{A}_{\text{low}}$ to the left, we know that the part $B_{\tilde{A}_{\text{up}}} \cap T_{\tilde{A}_{\text{low}}}$ of $B_{\tilde{A}_{\text{up}}}$ is not charged by any other hole, because it does not belong to the boundary of a hole, and the lid is defined uniquely.

We define points, P and P', for $\hat{H}_h$ in the same way as in the preceding section. Independent of $\hat{H}_h$'s type, $\tilde{A}'_{\text{up}}$ would get charged only for the rectangle R_1 induced by P, P', and N, as well as for the triangle below R_1 if we would use Lemma 12 and Corollary 2.

Next we show that we do not have to charge $\tilde{A}'_{\text{up}}$ for R_1 at all, because the part of R_1 that is above D_l^g is already included in the bound for $\hat{H}_g$, and the remaining part can be charged to $B_{\tilde{A}_{\text{up}}}$ and $R_{\tilde{A}_{\text{low}}}$. $\tilde{A}'_{\text{up}}$ will get charged only $\frac{1}{2}$ for the triangle.

D_l^g splits R_1 into a part that is above this line and a part that is below this line. The latter part of R_1 is not included in the bound for $\hat{H}_g$. Let F be the intersection of $\partial\hat{H}_g$ and D_l^g that caused the creation of $\hat{H}_h$. If $F \in R_{\tilde{A}_{\text{low}}}$, then this part is at most $\frac{1}{2}(\rho^h_{\text{low}})^2$, where ρ^h_{low} is the length of $\overline{P'F}$. We charge $\frac{1}{2}$ to $R_{\tilde{A}_{\text{low}}}$. If $F \in B_{\tilde{A}_{\text{up}}}$, then the part of R_1 below D_l^g can be split into a rectangular part of area $\rho^h_{\text{low}} \cdot \beta^h_{\text{up}}$, and a triangular part of area $\frac{1}{2}(\rho^h_{\text{low}})^2$; see Fig. 4.6. Here β^h_{up} is the length of $\overline{PF}$. The cost of the triangular part is charged to $R_{\tilde{A}_{\text{low}}}$. Note that $B_{\tilde{A}_{\text{up}}}$ exceeds $\tilde{A}_{\text{low}}$ to the left and to the right and that the part that exceeds $\tilde{A}_{\text{low}}$ to the right is not charged. Moreover, ρ^h_{low} is not larger than $B_{\tilde{A}_{\text{up}}} \cap T_{\tilde{A}_{\text{low}}}$, *i.e.,* the part of $B_{\tilde{A}_{\text{up}}}$ that was not charged before. Therefore, we can charge the rectangular part completely to $B_{\tilde{A}_{\text{up}}}$. Hence, $\tilde{A}'_{\text{up}}$ is charged $\frac{1}{2}$ for the triangle below R_1, and $\tilde{A}_{\text{up}}$ is charged at most 2.5 in total.

Holes Containing Parts of ∂S So far we did not consider holes whose boundary touches ∂S. We show in this section that these holes are just special cases of the ones discussed in the preceding sections.

Because the top side of a square never gets charged for a hole, it does not matter whether a part of B_S belongs to the boundary. Moreover, for any hole, $\hat{H}_h$, either L_S or R_S can be a part of $\partial\hat{H}_h$, because otherwise there exits a curve with one endpoint on L_S and the other endpoint on R_S, with the property that this curve lies completely inside of $\hat{H}_h$. This contradicts the existence of the bottom sequence of a square lying above the curve.

For a hole $\hat{H}_h$ with L_S contributing to $\partial\hat{H}_h$, we can use the same arguments as in the proof for Lemma 4 to show that $L_S \cap \partial\hat{H}_h$ is a single line segment. Let P be the topmost point of this line segment and $\tilde{A}_1$ be the

	Non-virtual Lid					Virtual Lid				Total
	Type I	Type II	L_S	R_S	Max.	Type I	Type II	R_S	Max.	
Left Side	0	$\frac{1}{2}$	$\frac{1}{2}$	0	$\frac{1}{2}$	0	0	0	0	$\frac{1}{2}$
Bottom Side	1	1	1	1	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1.5
Right Side	$\frac{1}{2}$	$\frac{1}{2}$	0	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	0	$\frac{1}{2}$
Total					2				$\frac{1}{2}$	2.5

Table 4.1: The charges to the different sides of a single square. Summing up the charges to the different sides, we conclude that every square gets a total charge of at most 2.5.

square containing P. $\tilde{A}_1$ must have a bottom neighbor, $\tilde{A}_k$, and $\tilde{A}_k$ must have a left neighbor, $\tilde{A}_{k-1}$, we get $P_{k,1} \in B_{\tilde{A}_1}$ and $P_{k-1,k} \in L_{\tilde{A}_k}$, respectively. We define the right diagonal, D_r, and the point Q' as above and conclude that $\hat{H}_h$ lies completely inside the polygon formed by $L_S \cap \partial\hat{H}_h$, D_r, and the part of $\partial\hat{H}_h$ that is between P and Q' in clockwise order. We split this polygon into a rectangle and a triangle in order to obtain charges of 1 to $B_{\tilde{A}_1}$ and $\frac{1}{2}$ to $L_{\tilde{A}_k}$.

Now, consider a hole where a part of R_S belongs to $\partial\hat{H}_h$. We denote the topmost point on $R_S \cap \partial\hat{H}_h$ by Q, and the square containing Q by $\tilde{A}_1$. We number the squares in counterclockwise order and define the left diagonal, D_l, as above. Now we consider the intersections of D_l and eliminate them by creating new holes. After this, the modified hole $\hat{H}_h^{(z)}$ can be viewed as a hole of Type II, for which the part on the right side of V_Q has been cut off; compare Corollary 2. We obtain charges of 1 to $B_{\tilde{A}_1}$ and $\frac{1}{2}$ to $R_{\tilde{A}_2}$. For the copy of a square we get a charge of $\frac{1}{2}$ to the bottom side.

4.2.4 Summing up the Charges

Altogether, we have the charges from Table 4.1. The charges depend on the type of the adjacent hole (Type I, II, touching or not touching the strip's boundary), but the maximal charge dominates the other ones. Moreover, the square may also serve as a virtual lid. The maximal charges from a hole with non-virtual lid and those from a hole with virtual lid sum up to a total charge of 2.5 per square. This proves our claim from the beginning:

$$\sum_{h=1}^{s} |H_h| \leq 2.5 \cdot \sum_{i=1}^{n+1} a_i^2 \,.$$

4.3 The Strategy *SlotAlgorithm*

In this section we analyze a different strategy for the strip packing problem with Tetris and gravity constraint. This strategy provides more structure on the generated packing, which allows us to prove an upper bound of 2.6154 on the asymptotic competitive ratio.

4.3.1 The Algorithm

Consider two vertical lines of infinite length going upwards from the bottom side of S and parallel to the left and the right side of S. We call the area between these lines a *slot*, the lines the *left boundary* and the *right boundary* of the slot, and the distance between the lines the *width* of the slot.

Now, our strategy *SlotAlgorithm* works as follows: We divide the strip S of width 1 into slots of different widths; for every $j = 0, 1, 2 \ldots$, we create 2^j slots of width 2^{-j} side by side, *i.e.,* we divide S into one slot of width 1, two slots of width 1/2, four slots of width 1/4, and so on. Note that a slot of width 2^{-j} contains 2 slots of width 2^{-j-1}; see Fig. 4.7.

For every square A_i, we round the side length a_i to the smallest number 2^{-k_i} that is larger than or equal to a_i. Among all slots of with 2^{-k_i}, we place A_i in the one that allows A_i to be placed as near to the bottom of S as possible, by moving A_i down along the left boundary of the chosen slot until another square is reached. The algorithm clearly satisfies the Tetris and the gravity constraints, and next we show that the produced height is at most 2.6154 times the height of an optimal packing.

4.3.2 Analysis

Let A_i be a square placed by the SlotAlgorithm in a slot T_i of width 2^{-k_i}. If $a_i \leq \frac{1}{2}$, we define δ_i as the distance between the right side of A_i and the right boundary of the slot of width 2^{-k_i+1} that contains A_i, and we define $\delta_i' = \min\{a_i, \delta_i\}$. We call the area obtained by enlarging A_i by δ_i' to the right and by $a_i - \delta_i'$ to the left (without A_i itself) the *shadow* of A_i and denote it by A_i^S. Thus, A_i^S is an area of the same size as A_i and lies completely inside a slot of twice the width of A_i's slot. If $a_i \geq \frac{1}{2}$, we enlarge A_i only to the right side and call this area the shadow. Moreover, we define the *widening* of A_i as $A_i^W = (A_i \cup A_i^S) \cap T_i$; see Fig. 4.7.

Now, consider a point P in S that is not inside an A_j^W for any square A_j. We charge P to the square, A_i, if A_i^W is the first widening that intersects the vertical line going upwards from P. We denote by F_{A_i} the set of all points charged to A_i and by $|F_{A_i}|$ its area. For points lying on the left or the

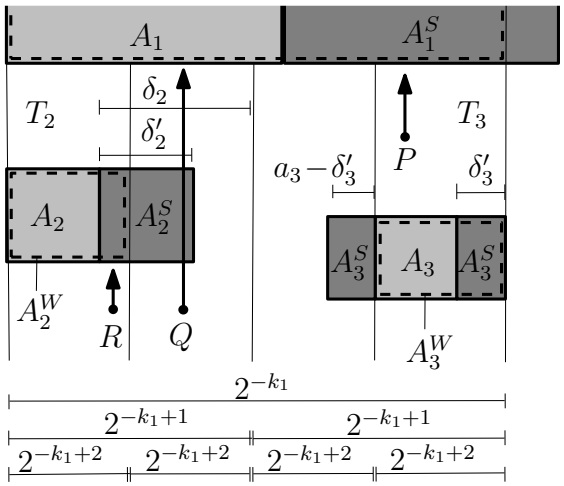

Figure 4.7: Squares A_i, $i = 1, 2, 3$, with their *shadows* A_i^S and their *widening* A_i^W. δ_2' is equal to a_2 and δ_3' is equal to δ_3. The points P and Q are charged to A_1. R is not charged to A_1, but to A_2.

right boundary of a slot, we break ties arbitrarily. For the analysis, we place a closing square, A_{n+1}, of side length 1 on top of the packing. Therefore, every point in the packing that does not lie inside an A_j^W is charged to a square. Because A_i and A_i^S have the same area, we can bound the height of the packing produced by the SlotAlgorithm by

$$2 \cdot \sum_{i=1}^{n} a_i^2 + \sum_{i=1}^{n+1} |F_{A_i}| \; .$$

Theorem 11. *The SlotAlgorithm is (asymptotically) 2.6154-competitive.*

Proof. The height of an optimal packing is at least $\sum_{i=1}^{n} a_i^2$, and therefore, it suffices to show that $|F_{A_i}| \leq 0.6154 \cdot a_i^2$ holds for every square A_i. We construct for every A_i a sequence of squares $\tilde{A}_1^i, \tilde{A}_2^i, \ldots, \tilde{A}_m^i$ with $\tilde{A}_1^i = A_i$ (to ease notation, we omit the superscript i in the following). We denote by E_j the extension of the bottom side of $\tilde{A}_j$ to the left and to the right; see Fig. 4.8.

We will show that by an appropriate choice of the sequence, we can bound the area of the part of $F_{\tilde{A}_1}$ that lies between a consecutive pair of extensions, E_j and E_{j+1}, in terms of $\tilde{A}_{j+1}$ and the slot width. From this we will derive the upper bound on the area of $F_{\tilde{A}_1}$. We assume throughout the proof that the square $\tilde{A}_j$, $j \geq 1$, is placed in a slot, T_j, of width 2^{-k_j}. Note that $F_{\tilde{A}_1}$ is completely contained in T_1.

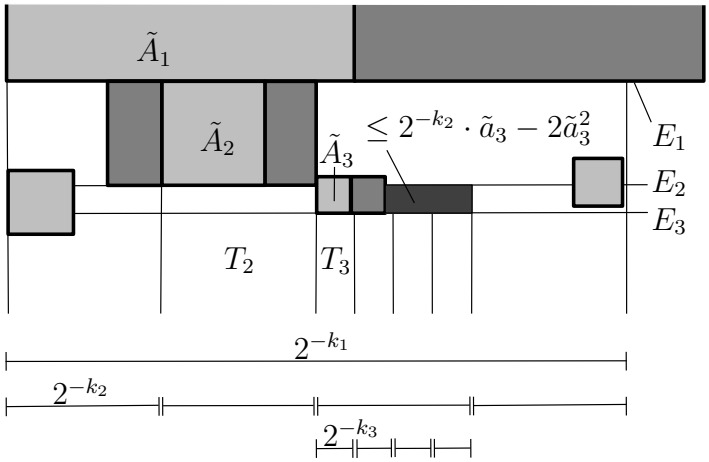

Figure 4.8: The first three squares of the sequence (light gray) with their shadows (gray). In this example, $\tilde{A}_2$ is the smallest square that bounds $\tilde{A}_1$ from below. $\tilde{A}_3$ is the smallest one that intersects E_2 in an active slot (w.r.t. E_2) of width 2^{-k_2}. There has to be an intersection of E_2 and some square in every active slot because, otherwise, there would have been a better position for $\tilde{A}_2$. T_2 is nonactive, (w.r.t. E_2) and of course, also w.r.t. all extension E_j, $j \geq 3$. The part of $F_{\tilde{A}_1}$ that lies between E_1 and E_2 has size $2^{-k_1}\tilde{a}_2 - 2\tilde{a}_2^2$.

A slot is called *active (with respect to E_j and $\tilde{A}_1$)* if there is a point in the slot that lies below E_j and that is charged to $\tilde{A}_1$ and *nonactive* otherwise. If it is clear from the context we leave out the $\tilde{A}_1$.

The sequence of squares is chosen as follows: $\tilde{A}_1$ is the first square and the next square, $\tilde{A}_{j+1}$, $j = 1, \ldots, m-1$, is chosen as the smallest one that intersects or touches E_j in an active slot (w.r.t. E_j and $\tilde{A}_1$) of width 2^{-k_j} and that is not equal to $\tilde{A}_j$; see Fig. 4.8. The sequence ends if all slots are nonactive w.r.t. to an extension E_m. We prove each of the following claims by induction:

Claim A: $\tilde{A}_{j+1}$ exists for $j+1 \leq m$ and $\tilde{a}_{j+1} \leq 2^{-k_j-1}$ for $j+1 \leq m$, *i.e.,* the sequence exists and its elements have decreasing side length.

Claim B: The number of active slots (w.r.t. E_j) of width 2^{-k_j} is at most

$$\begin{cases} 1 & \text{, for } j = 1 \text{ and} \\ \prod_{i=2}^{j} (\frac{1}{2^{k_i-1}} 2^{k_i} - 1) & \text{, for } j \geq 2\,. \end{cases}$$

Claim C: The area of the part of $F_{\tilde{A}_1}$ that lies in an active slot of width 2^{-k_j} between E_j and E_{j+1} is at most $2^{-k_j}\tilde{a}_{j+1} - 2\tilde{a}_{j+1}^2$.

Proof of Claim A: If $\tilde{A}_1$ is placed on the bottom of S, $F_{\tilde{A}_1}$ has size 0 and $\tilde{A}_1$ is the last element of the sequence. Otherwise, the square $\tilde{A}_1$ has at least one bottom neighbor, which is a candidate for the choice of $\tilde{A}_2$.

Now suppose for a contradiction that there is no candidate for the choice of the $(j+1)$th element. Let T' be an active slot in T_1 (w.r.t. E_j) of width 2^{-k_j} where E_j is not intersected by a square in T'. If there is an ε such that for every point, $P \in (T' \cap E_j)$, there is a point, P', at a distance ε below P which is charged to $\tilde{A}_1$, we conclude that there would have been a better position for $\tilde{A}_j$. Hence, there is at least one point, Q, below E_j that is not charged to $\tilde{A}_1$; see Fig 4.9. Consider the bottom sequence (as defined in Section 4.2.1) of the square Q is charged to. This sequence must intersect E_j outside of T' (by the choice of T'). This implies that one of its elements must intersect the left or the right boundary of T', and we can conclude that this square has at least the width of T'. This is because (by the algorithm) a square with rounded side length $2^{-\ell}$ cannot cross a slot's boundary of width larger than $2^{-\ell}$. In turn, a square with rounded side length larger than the width of T' completely covers T', and T' cannot be active w.r.t. to E_j and $\tilde{A}_1$. Thus, all points in T' below E_j are charged to this square; a contradiction. This proves that there is a candidate for the choice of $\tilde{A}_{j+1}$.

Suppose $\tilde{a}_2 > 2^{-k_1-1}$. Then, $\tilde{A}_2$ was placed in a slot of width at least 2^{-k_1}. Thus, its widening has width at least 2^{-k_1}, and $\tilde{A}_2$ is a bottom neighbor of $\tilde{A}_1$. Then, no point in T_1, below E_1, is charged to $\tilde{A}_1$, and hence, T_1 is nonactive w.r.t. E_1 and $\tilde{A}_1$. This implies, that $\tilde{A}_2$ does not belong to the sequence; a contradiction.

Because we chose $\tilde{A}_{j+1}$ to be of minimal side length, $\tilde{a}_{j+1} \geq 2^{-k_j}$ would imply that all slots inside T are nonactive (w.r.t. E_j). Therefore, if $\tilde{A}_{j+1}$ belongs to the sequence, $\tilde{a}_{j+1} \leq 2^{-k_j-1}$ must hold.

Proof of Claim B: Obviously, there is at most one active slot of width 2^{-k_1}; see Fig. 4.8. By the induction hypothesis, there are at most

$$(\frac{1}{2^{k_1}}2^{k_2}-1)\cdot(\frac{1}{2^{k_2}}2^{k_3}-1)\cdot\ldots\cdot(\frac{1}{2^{k_{j-2}}}2^{k_{j-1}}-1)$$

active slots of width $2^{-k_{j-1}}$ (w.r.t. E_{j-1}). Each of these slots contains $2^{k_j-k_{j-1}}$ slots of width 2^{-k_j}, and in every active slot of width $2^{-k_{j-1}}$ at least one slot of width 2^{-k_j} is nonactive because we chose $\tilde{A}_j$ to be of minimum side length. Hence, the number of active slots (w.r.t. E_j) is a factor of $(\frac{1}{2^{k_{j-1}}}2^{k_j}-1)$ times larger than the number of active slots (w.r.t. E_{j-1}).

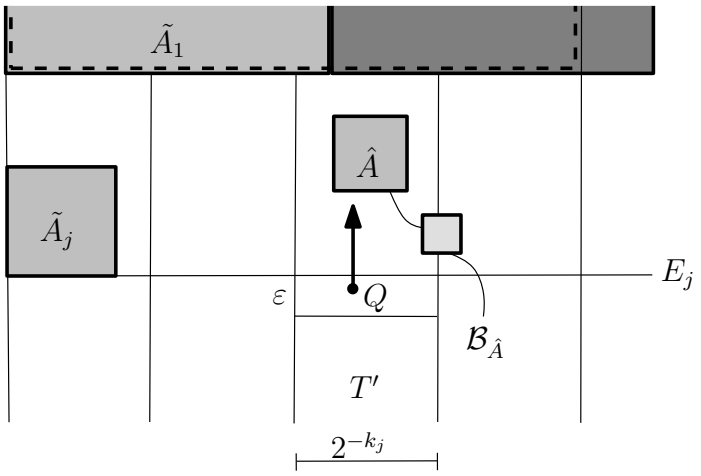

Figure 4.9: If E_j is not intersected in an active slot of size 2^{-k_j} we obtain a contradiction: Either there is a position for A_j that is closer to the bottom of S or there is a square that makes E_j nonactive. $\hat{A}$ is the square Q is charged to, $\mathcal{B}_{\hat{A}}$ its bottom sequence.

Proof of Claim C: The area of the part of $F_{\tilde{A}_1}$ that lies between E_1 and E_2 is at most $2^{-k_1}\tilde{a}_2 - 2\tilde{a}_2^2$; see Fig. 4.8. Note that we can subtract the area of $\tilde{A}_2$ twice, because $\tilde{A}_2^S$ was defined to lie completely inside a slot of width $2^{-k_2+1} \leq 2^{-k_1}$ and is of same area as $\tilde{A}_2$.

By the choice of $\tilde{A}_{j+1}$ and because in every active slot of width 2^{-k_j} there is at least one square that intersects E_j (points below the widening of this square are not charged to $\tilde{A}_1$) we conclude that the area of $F_{\tilde{A}_1}$ between E_j and E_{j+1} is at most $2^{-k_j}\tilde{a}_{j+1} - 2\tilde{a}_{j+1}^2$, in every active slot of width 2^{-k_j}.

Altogether, we proved that the sequence is well defined and we calculated an upper bound on the number of active slots and an upper bound on the size of the part of $|F_{\tilde{A}_1}|$ that is contained in an active slot. Multiplying the number and the size yields an upper bound on $|F_{\tilde{A}_1}|$ of

$$|F_{\tilde{A}_1}| \leq (\frac{\tilde{a}_2}{2^{k_1}} - 2\tilde{a}_2^2) \cdot 1 + \sum_{j=2}^{m} \left(\frac{\tilde{a}_{j+1}}{2^{k_j}} - 2\tilde{a}_{j+1}^2 \right) \prod_{i=2}^{j} \left(\frac{2^{k_i}}{2^{k_{i-1}}} - 1 \right) .$$

This expression is maximized if we choose $\tilde{a}_{i+1} = 1/2^{k_i+2}$, for $i = 1, \ldots, m$, *i.e.*, $k_i = k_1 + 2(i-1)$.

We get:

$$\begin{aligned}
|F_{\tilde{A}_1}| &\leq \frac{1}{2^{k_1+2}} \cdot \frac{1}{2^{k_1}} - 2 \cdot \left(\frac{1}{2^{k_1+2}}\right)^2 \\
&\quad + \sum_{j=2}^{m} \left[\frac{1}{2^{k_1+2(j-1)}} \cdot \frac{1}{2^{k_1+2j}} - 2\left(\frac{1}{2^{k_1+2j}}\right)^2\right] \prod_{i=1}^{j-1} \left(\frac{2^{k_1+2i}}{2^{k_1+2(i-1)}} - 1\right) \\
&= \frac{1}{2^{k_1+3}} + \sum_{j=2}^{m} \left[\frac{1}{2^{2k_1+4j-2}} - \frac{1}{2^{2k_1+4j-1}}\right] \cdot 3^{j-1} \\
&= \frac{1}{2^{k_1+3}} + \sum_{j=2}^{m} \frac{3^{j-1}}{2^{2k_1+4j-1}} \\
&= \frac{1}{2^{k_1+3}} + \sum_{j=1}^{m-1} \frac{3^{j}}{2^{2k_1+4j+3}} \\
&\leq \sum_{j=0}^{\infty} \frac{3^j}{2^{2k_1+4j+3}} \; .
\end{aligned}$$

The fraction $|F_{\tilde{A}_1}|/\tilde{a}_1^2$ is maximized, if we choose $\tilde{a}_1$ as small as possible, *i.e.,* $\tilde{a}_1 = 2^{-k_1-1} + \varepsilon$. We conclude:

$$\frac{|F_{\tilde{A}_1}|}{\tilde{a}_1^2} \leq \sum_{j=0}^{\infty} \frac{2^{2k_1+2} \cdot 3^j}{2^{2k_1+4j+3}} = \sum_{j=0}^{\infty} \frac{3^j}{2^{4j+1}} = \frac{1}{2} \cdot \sum_{j=0}^{\infty} \left(\frac{3}{16}\right)^j = \frac{8}{13} = 0.6153...$$

Thus,

$$|F_{A_i}| \leq 0.6154 \cdot a_i^2 \; .$$

□

4.4 Lower Bounds

The lower bound construction for online strip packing introduced by Galambos and Frenk [GF93] and later improved by van Vliet [Vli92] relies on an integer programming formulation and its LP-relaxation for a specific bin packing instance. This formulation does not take into account that there has to be a collision free path to the final position of the item. Hence, it does not carry over to our setting.

The best asymptotic lower bound, we are aware of, is $\frac{5}{4}$. It is based on two sequences which are repeated iteratively. We denote by A_i^k, $k = 1, \ldots, 5$ and $i = 1, 2, \ldots$, the k-th square of the sequence in the i-th iteration, and we

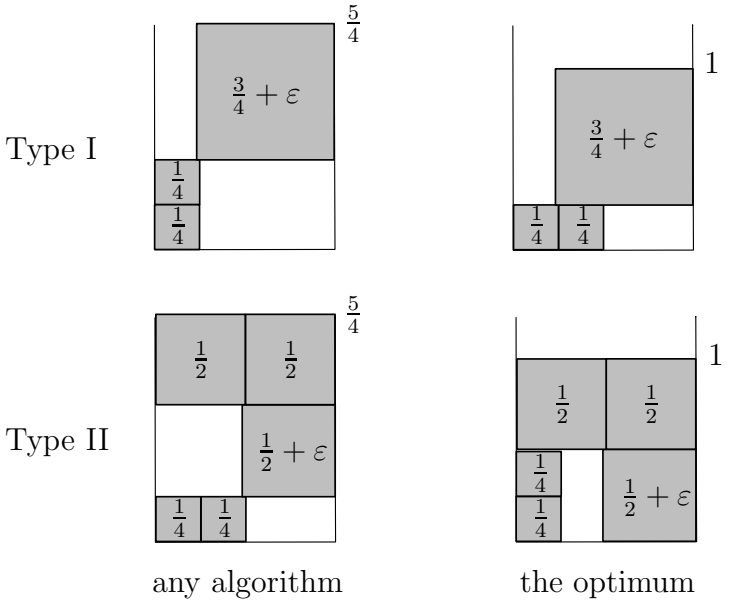

Figure 4.10: The left column shows possible packings of any algorithm for one iteration. The right column contains optimal packings. The top row displays the first and the bottom row the second type of sequence.

denote by H_i, $i = 1, 2, \ldots$, the height of the packing after the i-th iteration; we define $H_0 = 0$.

The first two squares of each sequence have a side length of $\frac{1}{4}$, that is, $a_i^1 = a_i^2 = \frac{1}{4}$. Now, depending on the choice of the algorithm, the sequence continues with one of the following two possibilities (see Fig. 4.10):

Type I: If the algorithms packs the first two squares on top of each other, with the bottom side of the lower square at height H_{i-1}, the sequence continues with a square of side length $\frac{3}{4} + \varepsilon$, *i.e.*, $a_i^3 = \frac{3}{4} + \varepsilon$ and $a_i^4 = a_i^5 = 0$ (upper left picture in Fig. 4.10).

Type II: Otherwise, the sequence continues with a square of side length $\frac{1}{2} + \varepsilon$ and two squares of side length $\frac{1}{2}$, *i.e.*, $a_i^3 = \frac{1}{2} + \varepsilon$ and $a_i^4 = a_i^5 = \frac{1}{2}$ (lower left picture in Fig. 4.10).

Lemma 13. *The height of the packing produced by any algorithm increases in each iteration, on average, by at least* $\frac{5}{4}$.

Proof. Consider the i-th iteration, $i = 1, 2, \ldots$. If the sequence is of Type I, the statement is obviously true because the square of side length $\frac{3}{4} + \varepsilon$ cannot pass any of the squares of side length $\frac{1}{4}$ which are packed on top of each other; see Fig. 4.10.

If the sequence is of Type II, we need to consider the previous iteration. If there was no previous iteration, then we know that A_i^1 and A_i^2 are both

placed on the bottom side of the strip. Because A_i^3 cannot be placed on the bottom side, and A_i^4 and A_i^5 cannot pass A_i^3, we get an increase of at least $\frac{5}{4}$.

If the sequence in the previous iteration was of Type I, H_{i-1} is determined by the square of side length $\frac{3}{4}+\varepsilon$. Hence, A_i^1 and A_i^2 are both placed on top of this square and the same arguments hold.

If the sequence in the previous iteration was of Type II, then either A_{i-1}^4 and A_{i-1}^5 are packed next to each other or on top of each other. In the first case, we can use the same arguments as in the case where there was no previous iteration. In the second case, A_{i-1}^4 and A_{i-1}^5 are placed an top of each other *and* on top of A_{i-1}^3, because they cannot pass a square with side length $\frac{1}{2}+\varepsilon$. This implies that, the last iteration contributed a height of at least $\frac{3}{2}$ to the height of the packing. No matter how the algorithm packs the squares from the current iteration (the first two squares might be placed at the same height or even deeper as the previous squares) it contributes a height of at least 1 to the packing. This proves an average increase of $\frac{5}{4}$ for both iterations. □

Theorem 12. *There is no algorithm with asymptotic competitive ratio smaller than $\frac{5}{4}$ for the online strip packing problem with Tetris and gravity constraint.*

Proof. The height of the packing produced by any algorithm increases by $\frac{5}{4}$ per iteration for the above instance (Lemma 13). The optimum can pack the squares belonging to one iteration always such that the height of the packing increases by at most 1; see the right column of Fig. 4.10. □

4.5 Conclusion

There are instances consisting only of squares for which the algorithm of Azar and Epstein does not undercut its proven competitive factor of 4. Hence, this algorithm is tightly analyzed. We proved competitive ratios of 3.5 and 2.6154 for BottomLeft and the SlotAlgorithm, respectively. Hence, both algorithms outperform the one by Azar and Epstein if the input consists only of squares.

Unfortunately, we do not know any instance for which BottomLeft produces a packing that is 3.5 times higher than an optimal packing. The best lower bound we know is $\frac{5}{4}$.

Moreover, we are not aware of an instance where the SlotAlgorithm reaches its upper bound of 2.6154. The instance consisting of squares with side length $2^{-k}+\delta$, for large k and small δ, gives a lower bound of 2 on the competitive ratio.

Hence, there is still room for improvement: Our analysis might be improved or there might be more sophisticated algorithms for the strip packing problem with Tetris and gravity constraint.

At this point, the bottleneck in our analysis for BottomLeft is the case where a square has large holes at the right, left, and bottom side and also serves as a virtual lid; see Fig. 4.2. This worst case can happen to only a few squares, but never to all of them. Thus, it might be possible to transfer charges between squares, which may yield a refined analysis. The same holds for the SlotAlgorithm and the sequence we constructed to calculate the size of the unoccupied area below a square.

In addition, it may be possible to apply better lower bounds on the packing than just the total area, *e.g.,* the one arising from *dual-feasible functions* by Fekete and Schepers [FS98].

Chapter 5

Point Sets with Minimum Average Distance

In this chapter we study two problems of selecting point sets such that the average L_1 distance between all pairs of points is minimized. The first problem considers the selection of $n \in \mathbb{N}$ points (a *town*) from the integer grid. In the second problem, for every number, n_i, from a given sequence, one has to select a shape of area n_i (a *city*) inside a fixed square, minimizing the interior distances.

5.1 Introduction

When looking at the map of a typical North American city (see Fig. 5.1), one is immediately convinced that the Euclidean distance between two points does not coincide with the actual walking distance. Indeed, the L_1 metric or Manhattan metric is a much more appropriate measure. Its value is exactly the length of any monotone and axis-parallel path between two locations. The point is that the cities have a regular structure, consisting of an allocation of a number of almost equal-sized squares (*city-blocks*).

Now, if one wants to design a new city from scratch, with a given number, n, of city-blocks, and the objective is to minimize the walking distance inside the city, one approach is to minimize the pairwise L_1 distances between the center points of the city blocks. In other words, we want to select n points from the integer grid (corresponding to the centers of the city-blocks) such that the sum of all pairwise L_1 distances between the points is minimized. Solutions for small values of n can be found in Fig. 5.2.

Besides the construction of new cities, which would indeed be a very rare application, the problems we study have some concrete applications in

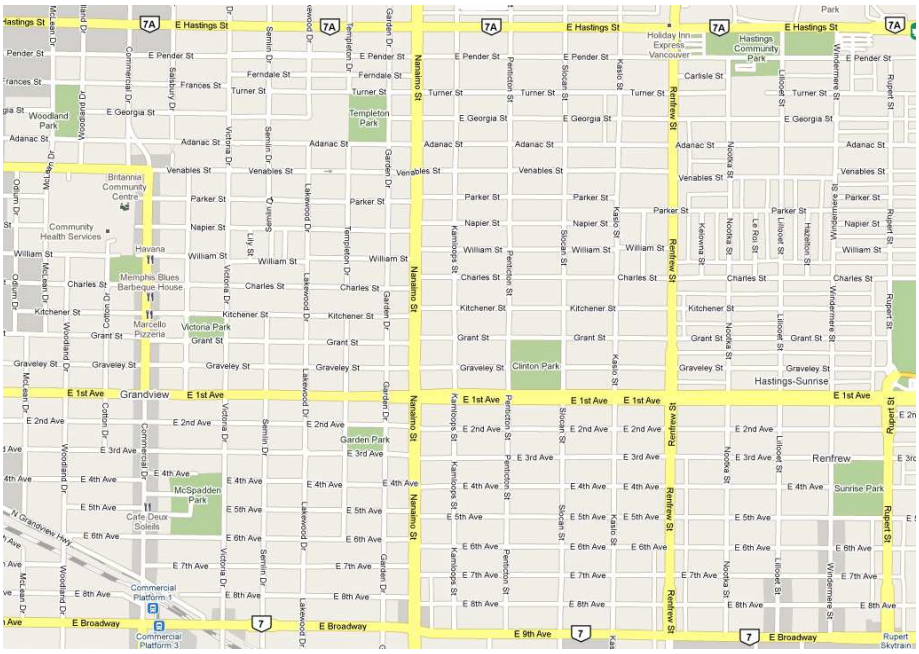

Figure 5.1: A part of the city map of Vancouver, showing the grid structure of the streets (image source: `www.maps.google.de`).

computer science, *e.g.,* in grid computing: Given a set of processors and a set of jobs, one has to select a subset of processors for every job. Processors, working on the same job, typically share information during runtime, causing communication overhead. This overhead depends in particular on the distances between the processors. Thus, if we want to minimize the communication cost, we should minimize the distances between all pairs of processors that work on the same job. If the processors are located on a two-dimensional integer grid (see, *e.g.,* [ML96, ML97] and [LAB+02]), that is, the distance between a pair of processors is their L_1 distance, this is exactly the problem described above.

In general, problems of this type, *e.g.,* the problem CLIQUE [GJ79], are hard even to approximate. However, in our setting we have additional geometric properties, so it is conceivable that more positive results can be achieved. However, even seemingly easy special cases are still surprisingly difficult. For example, for the special case of selecting one finite set of processors (the first problem studied in this chapter) such that the sum of all pairwise L_1 distances is minimized, there was no complexity result so far.

Indeed, for the shape of area 1 with minimum average pairwise L_1 distance (arising for the limit case of n approaching infinity), the shape can

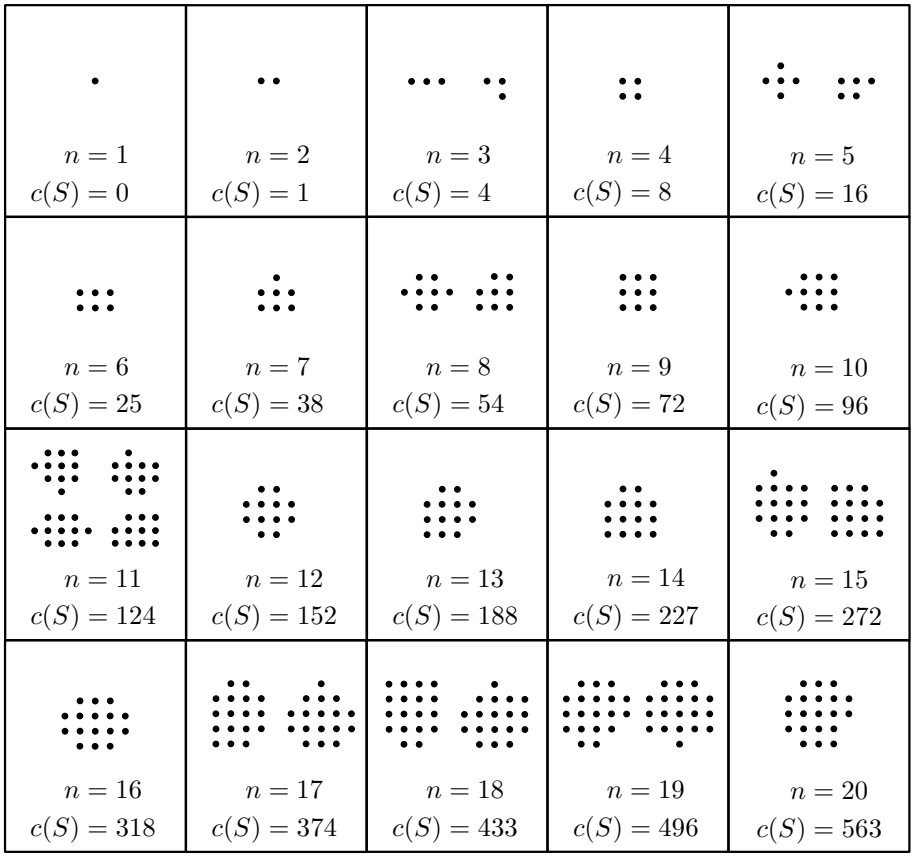

Figure 5.2: Optimal towns for $n = 1, \ldots, 20$. All optimal solutions are shown, up to symmetries; the numbers, $c(S)$, indicate the total distance between all pairs of points.

be described by a differential equation but no simple closed-form solution is known [BBDF04].

Even less is known for the problem of selecting k sets from a finite subset of grid points. This problem is the typical scenario in grid computing, as not only a single job but many of them have to be assigned to the processors. The second problem studied in this chapter considers the selection of k shapes inside a square such that the average L_1 distance between all pairs of points inside the shape are minimized. We do not consider the problem of selecting grid points but selecting points from the plane.

5.1.1 Our Results

We present the first positive result for the problem of choosing n points from the integer grid such that the sum of all pairwise L_1 distances is minimized,

by describing an $O(n^{7.5})$-time algorithm (Section 5.2). Our method is based on dynamic programming and (despite of its rather large exponent) for the first time allows computing optimal towns in time polynomial in n. Our implementation computes towns up to $n = 80$ in reasonable time. We published these results in [DFR+09].

Moreover, in Section 5.3 we consider the problem of choosing k shapes of area, n_i, $i = 1, \ldots, k$, and then packing them into a given square. Our goal is to minimize the maximum average pairwise L_1 distance inside the shape; where the maximum is taken over all shapes. We present a 5.3827-approximation algorithm for this problem.

5.1.2 Related Work

There are some problems closely related to our problem. We give an overview of the most important ones.

Scheduling of Malleable Tasks and Grid Computing Allocating jobs to processors from a given grid (grid computing) is known in the literature under the name of *scheduling of malleable tasks*. More precisely: Given a set of processors and a set of jobs, a processor can process one job at a time but a job might be distributed to more than one processor. The processing time of a job depends on the number of processors assigned to it; increasing the number of processors decreases the processing time, though not linearly in most models. The goal is to minimize the time until the last job is finished (the makespan). The cost for the communication between processors working on the same job is not explicitly considered. It is implicitly contained in the sublinear decrease of the processing time (when increasing the number of processors). Different scenarios (precedence constraints, preemption, and so on) have been considered; see Zhangs doctoral thesis [Zha04] and the references in it for a good starting point.

Mache and Lo [ML96, ML97] and Leung *et al.* [LAB+02] consider objective functions different from the one that aims at minimizing the makespan. They propose various metrics for measuring the quality of a processor allocation, including the average number of communication hops between processors. Leung *et al.* applied and evaluated their scheme based on space-filling curves on Cplant, a Sandia National Labs supercomputer[1]. They conclude that the average pairwise Manhattan distance between processors is an effective metric to optimize. Many other heuristics, algorithms, and measures for this problem have been proposed; see, *e.g.,* the references in [ML96].

[1] `www.sandia.gov`

The Continuous Version Motivated by the problem of storing records in a 2-dimensional array, Karp *et al.* [KMW75] study strategies that minimize average access time between successive queries; among other results, they describe an optimal solution for the continuous version of our problem: What shape of area 1 minimizes the average pairwise L_1 distance between two interior points? Bender *et al.* [BBDF04] solve this problem independently and introduce the term "city" for a selected shape. The optimal solution of this problem can be described by a differential equation, and no closed-form solution is known.

Selecting k points out of n Krumke *et al.* [KMN+97] consider the discrete problem of selecting a subset of k points from a set of n points minimizing their average pairwise distance. They prove a 2-approximation for metric distances and prove hardness of approximation for arbitrary distances. Bender *et al.* [BBD+08] solve the geometric version of this problem by describing a polynomial-time approximation scheme for minimizing the average Manhattan distance. For the reverse problem of *maximizing* the average Manhattan distance, see [FM03].

The k-median Problem and the Fermat-Weber Problem Given two sets of points, F and D, the *k-median problem* asks to choose a set of k points from F to minimize the average distance to the points in D. For $k = 1$, this is the classical *Fermat-Weber problem.*

Fekete *et al.* [FMW00, FMB05] consider the continuous version of this problem, *i.e.,* F and D are polygonal domains. They prove NP-hardness for general k and give efficient algorithms for some special cases.

The Quadratic Assignment Problem Our problem is a special case of the *quadratic assignment problem (QAP)*: Given n facilities, n locations, a matrix containing the amount of flow between any pair of facilities, and a matrix containing the distances between any pair of locations. The task is to assign every facility to a location such that the cost function, which is proportional to the flow between the facilities multiplied by the distances between the locations, is minimized. For a survey see [LAB+07]. The cost function in our problem and in the QAP are the same if we define the distances as the Manhattan distances between grid points and if we define all flows to be one. The QAP cannot be approximated within any polynomial factor, unless P=NP; see [SG76]. Hassin *et al.* [HLS09] consider the metric version of this problem with the flow matrix being a 0/1 incidence matrix of a graph. They state some inapproximability results as well as a constant-factor

approximation for the case in which the graph has vertex degree two for all but one vertex.

The Maximum Dispersion Problem The reverse version of the discrete problem, where the goal is to maximize the average distance between points, has also been studied: In the maximization version, called the *maximum dispersion problem*, the objective is to pick k points from a set of n points such that the pairwise distance is maximized. When the edge weights do not obey the triangle inequality, Kortsarz and Peleg [KP93] give an $O(n^{0.3885})$-approximation algorithm. Asahiro *et al.* [AITT00] improve this guarantee to a constant factor in the special case when $k = \Omega(n)$ and Arora *et al.* [AKK99] give a PTAS, when $|E| = \Omega(n^2)$ and $k = \Omega(n)$.

When the edge weights obey the triangle inequality, Ravi *et al.* [RRT94] give a 4-approximation that runs in $O(n^2)$ time and Hassin *et al.* [HRT97] give a 2-approximation that runs in $O(n^2 + k^2 \log k)$ time. For points in the plane and Euclidean distances, Ravi *et al.* [RRT94] give an approximation algorithm with performance bound arbitrarily close to $\pi/2 \approx 1.57$. For Manhattan distances, Fekete and Meijer [FM03] give an optimal algorithm for fixed k and a PTAS for general k. Moreover, they provide a $(\sqrt{2} + \varepsilon)$-approximation for Euclidean distances.

The Min-Sum k-Clustering Problem Another related problem is called *min-sum k-clustering* or *minimum k-clustering sum*. The goal is to separate a graph into k clusters such that the sum of pairwise distances between nodes in the same cluster is minimized. For general graphs, Sahni and Gonzalez [SG76] show that this problem is NP-hard to approximate within any constant factor for $k \geq 3$. In a metric space the problem is easier to approximate: Guttmann-Beck and Hassin [GBH98] give a 2-approximation, Indyk [Ind99] gives a PTAS for $k = 2$, and Bartel *et al.* [BCR01] give an $O(1/\epsilon \log^{1+\epsilon} n)$-approximation for general k.

5.2 Computing Optimal Towns

In this section we present an algorithm that selects n points from the integer grid minimizing the sum of the pairwise L_1 distances. First, we state some properties of optimal allocations, and after that we present our dynamic-programming approach.

5.2.1 Properties of Optimal Towns

We want to find a set of n distinct points from the integer grid $\mathbb{Z} \times \mathbb{Z}$ such that the sum of all pairwise L_1 distances is minimized. A set, $S \subset \mathbb{Z} \times \mathbb{Z}$, of cardinality, n, is an *n-town*. An n-town, S, is *optimal* if its *cost*

$$c(S) := \frac{1}{2} \cdot \sum_{s \in S} \sum_{t \in S} \|s - t\|_1 \tag{5.1}$$

is minimum. Fig. 5.2 shows optimal solutions for small n and their cost, Fig. 5.4 shows optimal solutions for $n = 58, 59, 60$, and Table 5.1 shows optimal cost values, $c(S)$, for $n \leq 80$. We define the *x-cost* $c_x(S)$ as $\sum_{\{s,t\} \in S \times S} |s_x - t_x|$, where s_x is the x-coordinate of s; *y-cost* $c_y(S)$ is the sum of all y-distances, and $c(S) = c_x(S) + c_y(S)$. For two sets, S and S', we define $c(S, S') = \sum_{\{s,s'\} \in S \times S'} \|s - s'\|_1$. A town, S, is *convex* if the set of grid points in the convex hull of S is equal to S.

In proving various properties of optimal towns, we will often make a local modification by removing a point, t, from the town and choosing a point, r, instead. The next lemma expresses the resulting cost change.

Lemma 14. *Let S be a town, $t \in S$ and $r \notin S$. Then,*

$$c((S \setminus t) \cup r) = c(S) - c(t, S) + c(r, S) - \|r - t\|_1 \ .$$

Proof. Let p be a point in S. Then, its distance to t is $\|t-p\|_1$ and the distance to r is $\|r-p\|_1$. Hence, the change in the cost function is $\|r-p\|_1 - \|t-p\|_1$. We need to subtract $\|r - t\|_1$ from the sum over all points in S because t is removed from S. □

Lemma 15. *An optimal n-town is convex.*

The following proof holds in any dimension and with any norm for measuring the distance between points.

Proof. Let S be an n-town which is not convex. Then, there are points, $x_1, x_2, \ldots, x_k \in S$, and a point, $x \notin S$, such that $x = \lambda_1 x_1 + \lambda_2 x_2 + \cdots + \lambda_k x_k$, for some $\lambda_1, \lambda_2, \ldots, \lambda_k \geq 0$ with $\sum \lambda_i = 1$. Because every norm is a convex function, and the sum of convex functions is again convex, the function, $f_S(x) = c(x, S) = \sum_{s \in S} \|x - s\|_1$, is convex. Therefore,

$$f_S(x) \leq \lambda_1 f(x_1) + \lambda_2 f(x_2) + \cdots + \lambda_k f(x_k),$$

which implies $f_S(x) \leq f_S(x_i)$ for some i. Using Lemma 14, we get

$$c((S \setminus x_i) \cup x) = c(S) - f_S(x_i) + f_S(x) - \|x - x_i\|_1 < c(S) \ ,$$

and, hence, a decrease in the cost function. Thus, S cannot be optimal. □

Obviously, if we translate every point from an n-town by the same vector, the cost of the town does not change. We want to distinguish towns because of their shape and not because of their position inside the grid, and therefore, we will only consider optimal towns that are placed around the origin. Lemma 16 makes this more precise: An optimal n-town is roughly symmetric with respect to a vertical and a horizontal symmetry line, see Fig. 5.3 for an illustration. Perfect symmetry is not possible since some rows or columns may have odd length and others even length.

We need some notation before: For an n-town, S, the i-th *column* of S is the set $C_i = \{\,(i,y) \in S : y \in \mathbb{Z}\,\}$ and the i-th *row* of S is the set $R_i = \{\,(x,i) \in S : x \in \mathbb{Z}\,\}$.

Lemma 16. *In every optimal n-town, S, the centers of all rows of odd length lie on a common vertical line, V_o. The centers of all rows of even length lie on a common line, V_e, that has distance $\frac{1}{2}$ from V_o. A corresponding statement holds for the centers of odd and even columns that lie on horizontal lines, H_o and H_e, of distance $\frac{1}{2}$. Moreover, without changing its cost, we can place S such that H_o and V_o are mapped onto the x-axis and y-axis, respectively, and H_e and V_e lie in the negative halfplanes.*

Proof. For a row, R_j, and $r \in \mathbb{Z}$, let $R_j + (r,0)$ be the row, R_j, horizontally translated by $(r,0)$. If two rows, R_i and R_j, are of the same parity, a straightforward calculation (using Lemma 14) shows that the cost $c(R_i, R_j + (r,0))$ is minimal if and only if the centers of R_i and $R_j + (r,0)$ have the same x-coordinate. If the parities differ, $c(R_i, R_j + (r,0))$ is minimized with centers having x-coordinates of distance $1/2$. The total cost of a town can be written as

$$c(S) = c_x(S) + c_y(S) = \sum_{i,j} \sum_{s \in R_i, t \in R_j} |s_x - t_x| + c_y(S)\ .$$

If we translate every row, R_i, of S horizontally by some $(r_i, 0)$, $c_y(S)$ does not change. The solutions that minimize $\sum_{s \in R_i, t \in R_j} |s_x + r_i - t_x + r_j|$ for all i, j simultaneously are exactly those that align the centers of all rows of even length on a vertical line, V_e, and the centers of all rows of odd length on a vertical line, V_o, at offset $\frac{1}{2}$ from V_e. The existence of the lines, H_o and H_e, follows analogously.

We can translate S such that H_o and V_o are mapped onto the x- and the y-axis and rotate it by a multiple of $90°$ degrees such that H_e and V_e lie in the negative halfplanes. These operations do not change $c(S)$. □

From the convexity statement in Lemma 15 (together with Lemma 16) we know that C_0 is the largest column, and the column lengths do not increase to both sides, and similarly for the rows. Our algorithm will only be based on this weaker property (*orthogonal convexity*); it will not make use of convexity *per se*. We will, however, use convexity one more time to prove that the lengths of the columns are $O(\sqrt{n})$, in order to bound the running time.

In the following, we assume that the symmetry property of the last lemma holds. For an n-town, S, let the *width* of S be $w(S) = \max_{i\in\mathbb{Z}} |R_i|$ and the *height* of S be $h(S) = \max_{i\in\mathbb{Z}} |C_i|$. We will now show that the width and the height cannot differ by more than a factor of 2. Together with convexity, this will imply that they are bounded by $O(\sqrt{n})$ (Lemma 18).

Lemma 17. *For every optimal n-town, S,*

$$w(S) > \frac{h(S)}{2} - 3 \text{ and } h(S) > \frac{w(S)}{2} - 3.$$

Proof. Let S be an n-town. For convenience we set $w = w(S)$, and $h = h(S)$. Suppose for a contradiction that $w \le h/2 - 3$. Let $t = (0, l)$ be the topmost and $(k, 0)$ be the rightmost point of S, with $l = \lfloor \frac{h-1}{2} \rfloor$ and $k = \lfloor \frac{w-1}{2} \rfloor$. Let $r = (k+1, 0)$. We show that $c((S \setminus t) \cup r) < c(S)$, and thus, S is not optimal. By Lemma 14, the change in the costs is $c(r, S) - c(t, S) - |k+l+1|$. We show that $c(r, S) - c(t, S) \le 0$ by calculating this difference column by column. This proves that replacing t with r yields a gain of at least $|l+k+1| \ge 1$, and we are done. Let us calculate the difference $c(r, C_j) - c(t, C_j)$ for a column, C_j, of height $|C_j| = s \le h$:

$$\begin{aligned}
c(r, C_j) - c(t, C_j) &= \sum_{i=-\lceil \frac{s-1}{2} \rceil}^{\lfloor \frac{s-1}{2} \rfloor} (|i| - (l - i)) + s \cdot (k + 1 - |j|) \\
&= \sum_{i=0}^{\lfloor \frac{s-1}{2} \rfloor} (i - (l - i)) + \sum_{i=1}^{\lceil \frac{s-1}{2} \rceil} (i - (l + i)) + s \cdot (k + 1 - |j|) \\
&= 2 \sum_{i=0}^{\lfloor \frac{s-1}{2} \rfloor} i - s \cdot l + s \cdot (k + 1 - |j|) \\
&\le \tfrac{(s-1)(s+1)}{4} - s \cdot \tfrac{h-2}{2} + s \cdot (k + 1) \le \tfrac{s^2}{4} - s \cdot \tfrac{h-2}{2} + s \cdot \tfrac{w+1}{2} \\
&= \tfrac{s}{4} \cdot (s - 2h + 2w + 6) \le \tfrac{s}{2} \cdot (-h + 2w + 6) \le 0
\end{aligned}$$

The last inequality follows because $w < \frac{h}{2} - 3$. □

Lemma 18. *For every optimal n-town we have*

$$\max\{w(S), h(S)\} \leq 2\sqrt{n} + 5.$$

Proof. Let $w = w(S)$ and $h = h(S)$. Assume without loss of generality that $h \geq w$. We know from Lemma 17 that $w > h/2 - 3$. By Lemma 16, we choose a topmost, a rightmost, a bottommost, and a leftmost point of S such that the convex hull of these four points is a quadrilateral with a vertical and a horizontal diagonal, approximately diamond-shaped. Let H be the set of all grid points contained in this quadrilateral. The area of the quadrilateral equals $(w-1)(h-1)/2$, and its boundary contains at least 4 grid points. Pick's theorem (compare [GO97]) says that the area of a simple grid polygon equals the number of its interior grid points, H_i, plus half of the number of the grid points, H_b, on its boundary minus 1. This implies $|H| = |H_\mathrm{i}| + |H_\mathrm{b}| = (|H_\mathrm{i}| + |H_\mathrm{b}|/2 - 1) + |H_\mathrm{b}|/2 + 1 \geq (w-1)(h-1)/2 + 3$. Because of Lemma 15, all points in H belong to S. Since H consists of at most n points, we have

$$n \geq |H| \geq \frac{(w-1)(h-1)}{2} + 3 > \frac{(\frac{h}{2} - 4)(h-1)}{2} + 3$$

Solving the equation $h^2 - 9h + 20 - 4n = 0$ shows that

$$h \leq 2\sqrt{n + \frac{1}{16}} + \frac{9}{2} \leq 2\sqrt{n} + 5\,.$$

□

5.2.2 Optimal Solutions

We will now describe a dynamic-programming algorithm for computing optimal towns.

We denote by $c_i = |C_i|$ the number of selected points in column i and by c_i^+ and c_i^- the row index of the topmost and bottommost selected point in C_i, respectively. We have $c_i = c_i^+ - c_i^- + 1$; see Fig. 5.3.

Lemma 19. *Let S be an optimal n-town containing the points (i, c_i^+) and (i, c_i^-). Then, all points inside the rectangle $[-i, i] \times [c_i^-, c_i^+]$ belong to S. The same holds for the points $(-i, c_{-i}^+)$, $(-i, c_{-i}^-)$, and the rectangle $[-i, i-1] \times [c_{-i}^-, c_{-i}^+]$.*

Proof. If (i, c_i^+) and (i, c_i^-) are contained in S then, by Lemma 16, $(-i, c_i^+)$ and $(-i, c_i^-)$ belong to S as well. By Lemma 15, all points inside the convex hull of these four points are contained in S. Fig. 5.3 shows an example for $i = 3$. The same arguments hold for the second rectangle. □

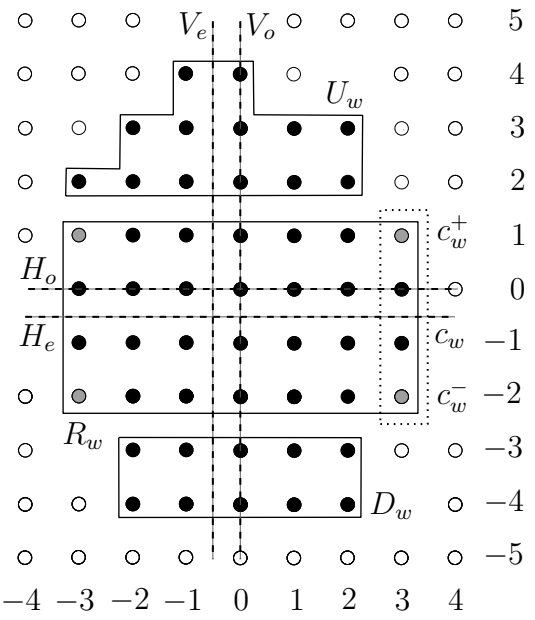

Figure 5.3: The lines, V_o, V_e, H_o, and H_e, from Lemma 16. The rectangle, R_w, and the set of points above and below it with cardinality, U_w and D_w, respectively. The gray points are the corner points of R_w. In this example, the height, c_w, of column w is $c = 4$.

Now we describe the dynamic program. It starts with the initial empty grid and chooses new columns alternating from the set of columns with nonnegative and with negative column index, *i.e.,* in the order $0, -1, 1, -2, 2, \ldots$. Let $w \geq 0$ be the index of the currently chosen column and fix c_w to a value c. We describe the dynamic program for columns with nonnegative index; columns with negative index are handled similarly.

From Lemma 19 we know that in every optimal solution, every point inside the rectangle, $R_w = [-w, w] \times [c_w^-, c_w^+]$, is selected. We define

$$\text{cost}(w, c, \Delta_w^{\text{UR}}, \Delta_w^{\text{DR}}, \Delta_w^{\text{UL}}, \Delta_w^{\text{DL}}, U_w, D_w)$$

as the minimum cost of a town with columns $-w, \ldots, w$ of height $c_i \geq c$ for $-w \leq i < w$ and $c_w = c$ where U_w points lie above the rectangle, R_w, having a total distance, Δ_w^{UL} and Δ_w^{UR}, to the upper-left and upper-right corner of R_w, respectively, and D_w points lie below R_w, having a total distance, Δ_w^{DL} and Δ_w^{DR}, to the lower-left and lower-right corner of R_w. For a given n, we are looking for the n-town with minimum cost where $(2w+1)c + U_w + D_w = n$. Next we show that $\text{cost}(w, c, \Delta_w^{\text{UR}}, \Delta_w^{\text{DR}}, \Delta_w^{\text{UL}}, \Delta_w^{\text{DL}}, U_w, D_w)$ can be computed recursively.

Consider the current column w with $c_w = c$. The cost from all points in this column to all points above R_w, in R_w, and below R_w can be expressed as

$$\sum_{k=c^-}^{c^+} (\Delta_w^{\mathrm{UR}} + (c^+ - k) \cdot U_w) + \sum_{i=-w}^{w} \sum_{j=c^-}^{c^+} \sum_{k=c^-}^{c^+} [(w-i) + |k-j|] + \sum_{k=c^-}^{c^+} (\Delta_w^{\mathrm{DR}} + (k - c^-) \cdot D_w) .$$

We can transform this sum into

$$c \cdot (\Delta_w^{\mathrm{UR}} + \Delta_w^{\mathrm{DR}} + U_w \cdot c^+ - D_w \cdot c^-) + c^- \cdot (D_w - U_w) \cdot ((c+1) \bmod 2) + \left(c^2 w + \frac{c^3 - c}{3}\right) \cdot (2w+1) - \frac{c^3 - c}{6}, \quad (5.2)$$

which, obviously, depends only on the parameters w, c, Δ_w^{UR}, Δ_w^{DR}, U_w, and D_w (the two parameters Δ_w^{UL} and Δ_w^{DL} are needed if we consider a column with negative index). We denote by $\mathrm{dist}(w, c, \Delta_w^{\mathrm{UR}}, \Delta_w^{\mathrm{DR}}, \Delta_w^{\mathrm{UL}}, \Delta_w^{\mathrm{DL}}, U_w, D_w)$ the expression (5.2) and state the recursion for the cost function:

$$\begin{aligned} &\mathrm{cost}(w, c, \Delta_w^{\mathrm{UR}}, \ldots, \Delta_w^{\mathrm{DL}}, U_w, D_w) \\ &\quad = \min_{c_{-w} \geq c} \{\mathrm{cost}(-w, c_{-w}, \Delta_{-w}^{\mathrm{UR}}, \ldots, \Delta_{-w}^{\mathrm{DL}}, U_{-w}, D_{-w})\} \\ &\qquad + \mathrm{dist}(w, c, \Delta_w^{\mathrm{UR}}, \ldots, \Delta_w^{\mathrm{DL}}, U_w, D_w) \quad (5.3) \end{aligned}$$

By Lemma 19, it suffices to consider only previous solutions with $c_{-w} \geq c$. In the step before, we considered the rectangle, $R_{-w} = [-w, w-1] \times [c_{-w}^-, c_{-w}^+]$. Hence, the parameters with index $-w$ can be computed from the parameters with index w as follows:

$$\begin{aligned} U_{-w} &= U_w - 2w \cdot (c_{-w}^+ - c^+), \\ D_{-w} &= D_w - 2w \cdot (c^- - c_{-w}^-), \\ \Delta_{-w}^{\mathrm{UR}} &= \Delta_w^{\mathrm{UR}} - \sum_{i=-w}^{w} \sum_{j=c^+ + 1}^{c_{-w}^+} \left[(w-i) + (j - c^+)\right] \\ &\qquad - [U_w - U_{-w}] \cdot (c_{-w}^+ - c^+ + 1), \\ \Delta_{-w}^{\mathrm{DR}} &= \Delta_w^{\mathrm{DR}} - \sum_{i=-w}^{w} \sum_{j=c^- - 1}^{c_{-w}^-} \left[(w-i) + (c^- - j)\right] \\ &\qquad - [D_w - D_{-w}] \cdot (c^- - c_{-w}^- + 1). \end{aligned}$$

The parameters, $\Delta^{\mathrm{UL}}_{-w}$ and $\Delta^{\mathrm{DL}}_{-w}$, can be computed analogously. The cost function is initialized as follows:

$$\mathrm{cost}(0,c,0,0,0,0,0,0) = \begin{cases} \frac{c^3-c}{6}, & \text{if } 0 \le c \le 2\sqrt{n}+5, \\ \infty, & \text{otherwise.} \end{cases}$$

The bound on c has been shown in Lemma 18 and $\frac{c^3-c}{6}$ is the cost of a single column of length c.

Theorem 13. *An optimal n-town can be computed by dynamic programming in $O(n^{15/2})$ time.*

Proof. We must fill an eight-dimensional array, $\mathrm{cost}(w, c, \Delta^{\mathrm{UR}}, \Delta^{\mathrm{DR}}, \Delta^{\mathrm{UL}}, \Delta^{\mathrm{DL}}, U, D)$. Let $C_{\max}$ denote the maximum number of occupied rows and columns in an optimum solution. From Lemma 18, we get $C_{\max} \in O(\sqrt{n})$.

The indices w and c range over an interval of size $C_{\max} = O(\sqrt{n})$. Let us consider a solution for some fixed w and c. The parameters U and D range between 0 and n. However, we can restrict the difference between U and D that we must consider: If we rotate the rectangle, $R = [-w, w] \times [c^-, c^+]$, around its horizontal symmetry axis, the U points above R and the D points below R will not match exactly, but in each column, they differ by at most one point, according to Lemma 16. It follows that $|U - D| \le C_{\max} = O(\sqrt{n})$. (If the difference is larger, such a solution can never lead to an optimal n-town, and hence, we do not need to explore those choices.) In total, we must consider only $O(n \cdot \sqrt{n}) = O(n^{3/2})$ pairs (U, D).

The same argument helps to reduce the number of quadruples, $(\Delta^{\mathrm{UL}}, \Delta^{\mathrm{UR}}, \Delta^{\mathrm{DL}}, \Delta^{\mathrm{DR}})$. Each Δ-variable can range between 0 and $n \cdot 2C_{\max} = O(n^{3/2})$. However, when rotating around the horizontal symmetry axis of R, each of the at most $D_{\max}$ differing points contributes at most $2C_{\max} = O(\sqrt{n})$ to the difference between the distance sums Δ^{UL} and Δ^{DL}. Thus, we have $|\Delta^{\mathrm{UL}} - \Delta^{\mathrm{DL}}| \le C_{\max} \cdot 2C_{\max} = O(n)$, and similarly, $|\Delta^{\mathrm{UR}} - \Delta^{\mathrm{DR}}| = O(n)$.

By a similar argument, rotating around the vertical symmetry axis of R, we conclude that $|\Delta^{\mathrm{UL}} - \Delta^{\mathrm{UR}}| = O(n)$ and $|\Delta^{\mathrm{DL}} - \Delta^{\mathrm{DR}}| = O(n)$. In summary, the total number of quadruples, $(\Delta^{\mathrm{UL}}, \Delta^{\mathrm{UR}}, \Delta^{\mathrm{DL}}, \Delta^{\mathrm{DR}})$, the algorithm must consider, is $O(n^{3/2}) \cdot O(n) \cdot O(n) \cdot O(n) = O(n^{9/2})$. In total, the algorithm processes $O(\sqrt{n}) \cdot O(\sqrt{n}) \cdot O(n^{3/2}) \cdot O(n^{9/2}) = O(n^7)$ 8-tuples. For each of the tuples, the recursion (5.3) must consider at most $C_{\max} = O(\sqrt{n})$ values c_{-w}, for a total running time of $O(n^{15/2})$. □

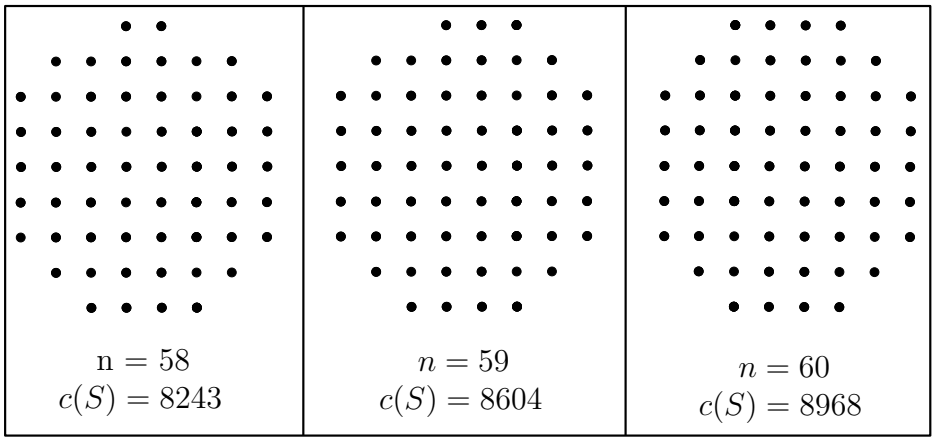

Figure 5.4: Optimal n-towns for $n = 58, 59, 60$.

5.2.3 Discussion

Our dynamic program allows the computation of optimal solutions for n up to 80 in reasonable time. Fig. 5.4 shows optimal solution for $n = 58, 59, 60$. These solutions are related in the sense that adding a point (in the top row) to the solutions for $n = 58$ yields an optimal solution for $n = 59$. The same holds for the optimal solutions for $n = 59$ and $n = 60$. This suggests that another approach might be to construct an optimal solution for n by adding a point on the boundary of an optimal $(n-1)$-town. However, this does not always work because, *e.g.,* the solution for $n = 9$ is a 3×3 square, that is not contained in the optimal solution for $n = 12$; see Fig. 5.2. It is not clear if this kind of construction might work for larger n, yielding a simple and fast algorithm for the construction of optimal towns.

Table 5.1 shows optimal solutions for $n = 1, \ldots, 80$. For some n there is more than one optimal solution, indicated by the number.

n	$c(S)$	n	$c(S)$	n	$c(S)$	n	$c(S)$
1	0	11	$124^{*(4)}$	21	632	31	1704
2	1	12	152	22	716	32	1840
3	$4^{*(2)}$	13	188	23	$804^{*(2)}$	33	1996
4	8	14	227	24	895	34	2153
5	$16^{*(2)}$	15	$272^{*(2)}$	25	992	35	2318
6	25	16	318	26	1091	36	2486
7	38	17	$374^{*(2)}$	27	1204	37	2656
8	$54^{*(2)}$	18	$433^{*(2)}$	28	1318	38	2847
9	72	19	$496^{*(2)}$	29	1442	39	3040
10	96	20	563	30	1570	40	3241

n	$c(S)$	n	$c(S)$	n	$c(S)$	n	$c(S)$
41	3446	51	5960	61	9354	71	$13700^{*(3)}$
42	3662	52	6248	62	$9749^{*(2)}$	72	14193
43	3886	53	6568	63	10146	73	14690
44	4112	54	6890	64	10556	74	15195
45	$4360^{*(2)}$	55	$7222^{*(2)}$	65	10972	75	15712
46	$4612^{*(2)}$	56	$7556^{*(2)}$	66	11400	76	16232
47	$4868^{*(2)}$	57	7896	67	$11836^{*(2)}$	77	16780
48	5128	58	8243	68	12280	78	17335
49	5398	59	8604	69	12728	79	$17904^{*(2)}$
50	5675	60	8968	70	13209	80	18478

Table 5.1: Optimal n-towns for n ranging from 1 to 80. $^{*(z)}$ indicates multiple solutions of cardinality z. Symmetric solutions are counted only once.

5.3 Packing Near-Optimal Cities

In this section we consider the problem of packing a set of *cities* into a unit square. A *city* is simply a shape of fixed area. We want to place the cities such that the maximum average L_1 distance between all pairs of points inside a city is minimized, where the maximum is taken over all cities. In other words, we want to find shapes and pack these shapes into a square such that the interior distances are small. Each shape is completely flexible and might even consist of two or more unconnected parts. The total size of all cities does not exceed the size of the given square. We present an approximation algorithm for this problem.

5.3.1 Problem Statement and Definitions

We are given a square, L, with side length 1 and a sequence, S, of numbers, $n_1, n_2, \ldots, n_k$, such that $\sum_{j=1}^{k} n_j \leq 1$. For every $j = 1, \ldots, k$, we want to select a city, N_j, of size, n_j, such that

$$\max_{1 \leq j \leq k} \left\{ \frac{1}{n_j^{2.5}} \iiiint_{(x,y),(u,v) \in N_j} |x-u| + |y-v| \, \mathrm{d}x \, \mathrm{d}y \, \mathrm{d}u \, \mathrm{d}v \right\}$$

is minimized, *i.e.,* we want to minimize the maximum average L_1 distance. We choose this scaled measure to preserve consistency with the papers [BBDF04] and [DFR+09]. Note that every distance between a pair of points is counted twice. Scaling the integral by a factor of $n^{-2.5}$ yields a dimensionless measure; compare [BBDF04].

For a city, N, we define

$$I[N] = \iiiint_{(x,y),(u,v)\in N} |x-u| + |y-v| \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}u\,\mathrm{d}v \;,$$

i.e., $I[N]$ is twice the L_1 distance between all pairs of points in N. Moreover we define, for $j = 1, \ldots, k$,

$$\phi_j = \frac{I[N_j]}{n_j^{2.5}} \;.$$

We call ϕ_j the *ϕ-value* (of the city N_j). For example for a city being a $w \times h$-rectangle we obtain a ϕ-value of

$$\frac{\frac{1}{3}(h+w)(w\cdot h)^2}{(h\cdot w)^{2.5}} \;.$$

If $w = h$ this expression is equal to $\frac{2}{3}$.

A point $p \in L$ is called *occupied* if it belongs to a city; otherwise it is called *unoccupied*. An area is unoccupied if all points in it are unoccupied and occupied if all points in it are occupied. The *aspect ratio* of a $w \times h$-rectangle is the maximum of $\frac{w}{h}$ and $\frac{h}{w}$.

Evidence for NP-hardness Unfortunately, we weren't able to prove NP-hardness for this problem. However, we provide some evidence that the problem is indeed hard. More precisely, we present two NP-hard problems which are related to our problem.

There is a strong need to control the shapes of the cities that are packed. For example long and thin cities lead, obviously, to large ϕ-values. Whereas compact forms such as a square or a circle lead to almost optimal ϕ-values. It straightforward to compute a ϕ-value of $\frac{2}{3}$ if the city is a square and a ϕ-value of 0.650403 for a circular city. The optimal value is 0.650245952951; see [BBDF04]. Thus, squares and circles are near optimal and it is convenient to choose one of these shapes for the cities. If one wants to pack all cities as squares, the problem reduces to the question whether a set of small squares can be packed into a large one. For orthogonal packings Leung *et al.* [LTW$^+$90] showed that this problem is NP-hard.

Moreover, there is some evidence for NP-hardness from the discrete version of the problem: choosing n_j cells from a *polyomino,* for each number n_j in the sequence. A *polyomino* consists of squares of equal size arranged with coincident sides. If the polyomino is allowed to have holes and the sequence consists of 3's only, then it is NP-hard to decide whether all shapes can be chosen such that they are optimal [DD07]. The reduction is made from planar 3-SAT.

Finally, if one wants to decide whether all cities can be packed such that they all have the optimal ϕ-value, this would either require to come up with a new solution for the problem of a single optimal city or to pack the cities with the shape computed in [BBDF04]. This shape is described by a differential equation with no known closed-form solution. Thus, packing cities with optimal ϕ-values would require to pack objects without explicitly knowing their boundary. This leads us to:

Conjecture 1. *Packing k cities such that the maximum ϕ-value is minimized is* NP*-hard.*

5.3.2 An Algorithm

Now we turn to the description of our approximation algorithm. For every $j = 1, \ldots, k$, we define

$$w_j = 2^{-r}, r \in \mathbb{N} \text{ such that } \frac{1}{4}w_j \leq \frac{n_j}{w_j} < w_j \ , \tag{5.4}$$

i.e., we can choose N_j to be a rectangle of width w_j and height $h_j := n_j/w_j$ such that h_j is larger than or equal to $w_j/4$ and smaller than w_j. Our algorithm works as follows:

We divide L into vertical *slots* of width 2^{-r}, for $r = 0, 1, ...$, and height 1. That is, we create one slot of width 1, two slots of width one half, and so on. Slots of the same width lie side by side, *i.e.,* they are pairwise disjoint. Note that a slot of width 2^{-r} is either completely inside or completely outside of a slot of width $2^{-r'}$, for $r \geq r'$. A slot of width w_j is called a w_j-slot.

We proceed in rounds and delete the numbers, placed in each round, from the sequence; initially, we consider the whole sequence. We number the elements which are currently in S by increasing size: $n_s, n_{s+1}, \ldots, n_l$. We choose the largest element, n_l, and the third largest element, $n_t := n_{l-2}$, from S. We consider the leftmost w_l-slot, S_l, that is not completely filled yet. The city N_l will be placed as the last element inside the slot and we call N_l the *lid element*.

The city N_t is divided into two parts: the *inner part,* N_t^{in}, is placed inside S_l and the *outer part,* N_t^{out}, in the w_t-slot directly to the right of S_l; this slot will be contained in the slot chosen in the next iteration. We call N_t the *transition element*. Thus, $N_{t'}^{\text{out}}$, the outer part of the previously chosen transition element, $N_{t'}$, is already placed inside S_l. $N_{t'}^{\text{out}}$ is a rectangle of width $w_{t'}$ either placed in the lower left or the upper left corner of S_l. Without loss of generality, we assume that it is placed in the lower left corner; otherwise, we flip L horizontally and proceed in the same way; see Fig. 5.5.

Let F be the total unoccupied area in S_l (at the time when only $N_{t'}^{\text{out}}$ is placed in S_l) and let f be the size of F. We distinguish three cases:

Case A: $n_l + n_t + \sum_{j=s}^{t-1} n_j > f$,
i.e., there are enough elements not larger than n_t that fill together with n_t and n_l the slot S_l completely. Note that there is no N_j with $w_j = 1$ in this case because, otherwise, $\sum_{j=1}^{k} n_j > 1$. We choose $e \geq 0$ such that

$$n_l + n_t + \sum_{j=s}^{s+e-1} n_j < f \leq n_l + n_t + \sum_{j=s}^{s+e} n_j \ .$$

We call the elements n_i, $i \in \{s, s+1, \ldots, s+e\}$ *inside elements*. First we place the transition element, then the inside elements, and at last the lid element:

Transition element: We define the size of the inner and the outer part of N_t: $n_t^{\text{out}} = n_l + n_t + \sum_{j=s}^{s+e} n_j - f$, *i.e.,* the part of n_t not fitting into S_l. Moreover, $n_t^{\text{in}} = n_t - n_t^{\text{out}}$. We always choose N_t^{in} as a $w_t \times h_t^{\text{in}}$-rectangle where $h_t^{\text{in}} := n_t^{\text{in}}/w_t$ and N_t^{out} as a $w_t \times h_t^{\text{out}}$-rectangle where $h_t^{\text{out}} := n_t^{\text{out}}/w_t$.

If $w_l = w_{t'}$, then N_t^{in} is placed in the upper right corner of S_l and N_t^{out} is placed at the top of the w_t-slot which is on the right side of S_l (the left boundary of this slot and the right boundary of S_l coincide); see Fig. 5.5.

If $w_l/2 \geq w_{t'}$, then N_t^{in} is placed in the lower right corner of S_l and N_t^{out} is placed at the bottom of the w_t-slot which is on the right side of S_l (the left boundary of this slot and the right boundary of S_l coincide).

Inside elements: To ease the analysis we temporarily (until all inside elements and the lid element are placed) flip L horizontally if $w_l = w_{t'}$, because now we can pack the inside elements in the same as if $w_l/2 \geq w_{t'}$, *i.e.,* we can start at the side of L, that is now at the bottom, in both cases. Moreover, this ensures that the upper boundary of the area, used for the lid element, is a straight line. We use this in the analysis of lid elements. Let n_i be an inside element. We choose N_i as a $w_i \times h_i$-rectangle. Now, we consider all w_i-slots inside S_l and choose the one such that the bottom side of N_i can be placed as close to the bottom as possible. The inside elements are packed by decreasing size, *i.e.,* in the order $N_{s+e}, N_{s+e-1}, \ldots, N_s$.

Lid element: After placing the transition element and the inside elements, we choose the remaining unoccupied area in S_l for the lid element. By our construction, this part has exactly size n_l; hence, S_l is filled completely.

Case B: $n_l + n_t + \sum_{j=s}^{t-1} n_j \leq f$ and even $n_l + n_t + \sum_{j=s}^{t-1} n_j + n_{l-1} \leq f$,
i.e., all remaining elements fit into the slot S_l. Then, n_l is the lid element and all other elements are inside elements; there is no transition element. The area for N_l is obtained by moving down a horizontal line from the top of S_l, until there is unoccupied space of size n_l above the line.

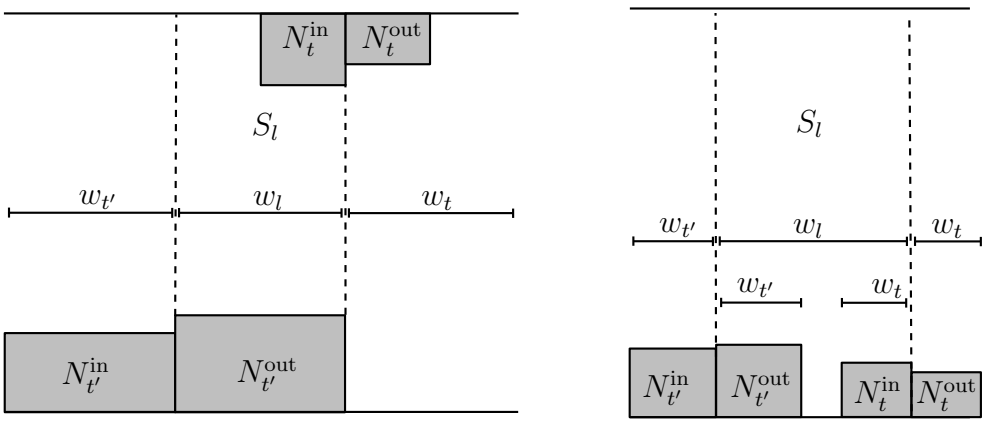

Figure 5.5: The two choices for the transition element. In the left picture $w_{t'} = w_l$. In this situation we flip L horizontally. Thus, in both situations there is a straight line at the top of S_l (either the boundary of L or the bottom side of $N_{t'}^{\text{out}}$). N_t^{in} and $N_{t'}^{\text{out}}$ never intersect.

Case C: $n_l + n_t + \sum_{j=s}^{t-1} n_j \leq f$ and $n_l + n_t + \sum_{j=s}^{t-1} n_j + n_{l-1} > f$, *i.e.,* $N_s, \ldots, N_{t-1}, N_t, N_l$ fit into S_l but $N_s, \ldots, N_{t-1}, N_t, N_l$ *and* N_{l-1} do not. Then, N_l is the lid element and $N_s, N_{s+1}, \ldots, N_t$ are inside elements. The area for N_l is obtained in the same way as in Case B; N_{l-1} is the lid element in the next iteration.

After placing the transition element, the inside elements and the lid element, we delete these numbers from S. We continue if S is not empty.

5.3.3 Analysis

From now on S_l is the current slot, N_l the lid element, N_t the transition element, N_i an inside element, and $N_{t'}$ the transition element from the previous iteration. We assume throughout this section that $N_{t'}$ is placed at the bottom of S_l because, otherwise, we can flip L and proceed in the same way. Next we prove the correctness and the performance guarantee for the algorithm. We start with the correctness of the choice of the slot in every iteration:

Lemma 20. *Let S be non-empty, S_l the slot in the current iteration, and $S_{l'}$ the slot from the last iteration. Then,*

(i) there is a slot of the same width as $S_{l'}$ to the right of $S_{l'}$, and

(ii) S_l can be chosen as described in the algorithm.

Proof. (i) Obviously, we can choose a slot of width $w_l \leq 1$ in the first iteration. Let R be the right boundary of $S_{l'}$, and d the distance between L's left side and R.

If not all elements from the sequence S where packed into $S_{l'}$ (*i.e.,* $S_{l'}$ was processed according to Case A or C), then $\sum_{j=1}^{k} n_j \geq d$. This means that not all N_j's, $j = 1, \ldots, k$, can be placed on the left side of R. Hence, if R would coincide with the right side of L, we get a contradiction to $\sum_{j=1}^{k} n_j \leq 1$. Otherwise, by the construction of the slots, there is at least one slot of the same width as $S_{l'}$ to the right of $S_{l'}$.

(ii) By our choice, the width of the slot S_l is at most the width of $S_{l'}$. □

It follows from the previous lemma that there is a suitable slot for the outer part of the transition element. This enables us to prove an upper bound on the ϕ-value for transition elements.

Lemma 21. *Let N_t be a transition element. Then,*

(i) N_t can be chosen as described in the algorithm, and

(ii) $\phi_t \leq 3$.

Proof. (i) We start with the inner part of N_t. If there is a transition element, $N_{t'}$, then $w_{t'}, w_t \leq 1/2$, because there are at least two slots. This implies $h_{t'}^{\text{out}}, h_t \leq 1/2$ (because of Eq. 5.4). Hence, if N_t^{in} is placed at the top of S_l, there is no intersection between N_t^{in} and $N_{t'}^{\text{out}}$. If N_t^{in} is placed at the bottom of S_l, then $w_t, w_{t'} \leq w_l/2$, and again, there is not intersection between N_t^{in} and $N_{t'}^{\text{out}}$; compare Fig. 5.5.

We turn to the outer part. If there is a transition element, then not all elements from S fit into S_l. It follows from Lemma 20(ii) that there is a slot of width at least w_t to the right of S_l. Thus, we can choose N_t^{out} as a $w_t \times h_t^{\text{out}}$-rectangle and place it in the lower right corner of that slot.

(ii) N_t consists of two rectangles, N_t^{in} and N_t^{out}, lying side by side with the bottom side on the same horizontal line. For convenience we define $g := h_t^{\text{out}}$ and $f := h_t^{\text{in}}$, and w.l.o.g., we assume that $f \geq g$. We compute the integral $I[N_t^{\text{in}} \cup N_t^{\text{out}}]$ and get:

$$\begin{aligned}
\phi_t &\leq \max_{w_t,f,g} \left\{ \frac{I[N_t^{\text{in}} \cup N_t^{\text{out}}]}{[w_t(f+g)]^{2.5}} \right\} \qquad (5.5)\\
&= \max_{w_t,f,g} \left\{ \frac{\frac{1}{3}w_t^2(f^3 - 3fg(g-2w_t) + f^2(3g+w_t) + g^2(3g+w_t))}{[w_t(f+g)]^{2.5}} \right\}\\
&= \max_{w_t,f,g} \left\{ \frac{\frac{1}{3}w_t^2(f^3 + 4fgw_t - 9fg^2 + (3g+w_t)(f^2+2fg+g^2))}{[w_t(f+g)]^{2.5}} \right\}\\
&\leq \max_{w_t,f,g} \left\{ \frac{\frac{1}{3}w_t^2(f^3 + fg(4w_t-8g) + (3g+w_t)(f+g)^2)}{[w_t(f+g)]^{2.5}} \right\}
\end{aligned}$$

$$
\begin{aligned}
&\leq \max_{w_t,f,g}\left\{\frac{\frac{1}{3}w_t^2(f(f+g)^2+(f+g)^2(4w_t-8g)+(3g+w_t)(f+g)^2)}{[w_t(f+g)]^{2.5}}\right\}\\
&= \max_{w_t,f,g}\left\{\frac{\frac{1}{3}(f+4w_t-8g+3g+w_t)}{[w_t(f+g)]^{0.5}}\right\}\\
&= \max_{w_t,f,g}\left\{\frac{\frac{1}{3}(f+5w_t-5g)}{[w_t(f+g)]^{0.5}}\right\} \qquad (5.6)
\end{aligned}
$$

To find the maximum we distinguish four cases.

First case $g = \frac{w_t}{4}$: Then $\frac{w_t}{4} \leq f \leq \frac{3}{4}w_t$, because $g \leq f$ and $f + g \leq w_t$. From (5.6) we get:

$$
\begin{aligned}
\phi_t &\leq \max_{w_t}\left\{\frac{1}{3}\cdot\frac{\frac{3}{4}w_t+5w_t-\frac{5}{4}w_t}{\left[w_t(2\cdot\frac{w_t}{4})\right]^{0.5}}\right\}\\
&= \frac{1}{3}\cdot\frac{9}{2}\cdot\sqrt{2}\\
&\leq 2.1214
\end{aligned}
$$

Second case $g = f$: We have $\frac{w_t}{4} \leq 2f \leq w_t$, which is equivalent to $\frac{w_t}{8} \leq f \leq \frac{w_t}{2}$. We set $f = \lambda \cdot w_t$, for $\lambda \in [\frac{1}{8}, \frac{1}{2}]$, and we get from (5.6):

$$
\begin{aligned}
\phi_t &\leq \max_{w_t,\lambda}\left\{\frac{1}{3}\cdot\frac{\lambda\cdot w_t+5w_t-5\lambda\cdot w_t}{[w_t(2\lambda\cdot w_t)]^{0.5}}\right\}\\
&\leq \max_{\lambda}\left\{\frac{1}{3}\cdot\frac{5-4\lambda}{\sqrt{2\lambda}}\right\}
\end{aligned}
$$

The first derivative of the function inside the brackets is equal to zero for $\lambda = -\frac{5}{4}$, which is not contained in the interval $[\frac{1}{8}, \frac{1}{2}]$. Thus, the function attains its maximum at one of the boundaries of the interval $[\frac{1}{8}, \frac{1}{2}]$. We compute a value of 3 for $\lambda = \frac{1}{8}$ and a value of 1 for $\lambda = \frac{1}{2}$. Hence, we obtain a maximal value of 3.

Third case $g < f$ and $g < \frac{w_t}{4}$: If we increase g by ε and decrease f by ε the denominator of (5.5) stays the same. We claim that we increase the nominator. We compute $I[M_t^{\text{in}} \cup M_t^{\text{out}}]$ where M_t^{in} has height $f-\varepsilon$ and M_t^{out} has height $g+\varepsilon$, and we get:

$$
\begin{aligned}
&I[M_t^{\text{in}} \cup M_t^{\text{out}}] - I[N_t^{\text{in}} \cup N_t^{\text{out}}]\\
&= \frac{2}{3}\cdot w_t^2\cdot\varepsilon\cdot(4\varepsilon^2+\varepsilon\cdot(9g-3f-2w_t)-2(f-g)(3g-w_t))
\end{aligned}
$$

If the last summand inside the brackets, $-2(f-g)(3g-w_t)$, is positive, we can find an $\varepsilon > 0$ such that the whole expression is positive. Because $f \geq g$ and $g < \frac{w_t}{4}$, this is clearly fulfilled. Hence, we always increase the value if we increase g and decrease f. We can increase g until it reaches f or $\frac{w_t}{4}$, and the result follows from one of the previous cases.

Fourth case $g \geq \frac{w_t}{4}$: Then, $4g \geq w_t$. Moreover, we have $f + g \leq w_t$ and $f \geq g$. We get from (5.6):

$$\begin{aligned}
\phi_t &\leq \max_{f,g} \left\{ \frac{\frac{1}{3}(f + 20g - 5g)}{f+g} \right\} \\
&\leq \max_{f,g} \left\{ \frac{\frac{1}{3}(f + 15g)}{f+g} \right\} \\
&\leq \max_{f,g} \left\{ \frac{\frac{1}{3} \cdot 8 \cdot (f+g)}{f+g} \right\} \\
&= \frac{8}{3}
\end{aligned}$$

Hence, it holds in all four cases that $\phi_t \leq 3$. □

Now we turn to the analysis of the inside elements and the lid element. Mainly, we will show that placing the inside elements as rectangles is a feasible solution and finally, leads to an unoccupied area at the top of the slot; this area is then chosen for the lid element.

It is obviously true that there is no unoccupied area below N_t^{in}, because it is placed on the bottom side of L (recall that we rotate L if $w_l = w_{t'}$). Moreover, if $w_l/2 \geq w_{t'}$, there is also no unoccupied area below $N_{t'}^{\text{out}}$. The same holds for inside elements.

Lemma 22. *Let N_i be an inside element, $N_{t'}^{\text{out}}$, N_t^{in}, $N_{s+e}, N_{s+e-1}, \ldots, N_i$ already be placed in S_l, and let B_i be the extension of the bottom side of N_i. Then, the area below B_i in S_l is completely occupied.*

Proof. We need some properties of the packing first. Consider the situation before an inside element, N_j, $j \geq i$, is placed. We sweep down a horizontal line from the bottom side of $N_{t'}^{\text{out}}$ (if $w_l = w_{t'}$) or the top side of L (otherwise). We claim that the polygonal curve obtained from the sweep in every w_j-slot is a straight line. For a fixed w_j-slot, consider the topmost point on the corresponding polygonal curve and let N_p be the rectangle containing the point. Then, the point belongs to N_p's top side, and because, $w_p \geq w_j$ the top side of N_p ranges (at least) from the left to the right side of the w_j-slot. Thus, the polygonal curve is a straight horizontal line.

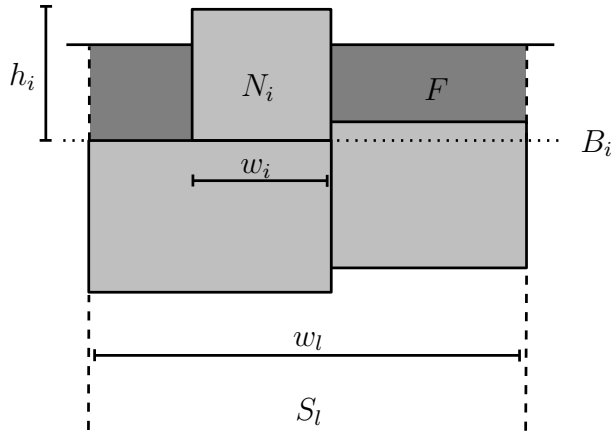

Figure 5.6: The situation where N_i's top side is above L's top side. We get a contradiction because the unoccupied area F (dark-gray) is smaller than n_l, and because, the area below B_i is completely occupied.

The rectangle, N_j, is placed in the w_j-slot with the straight line being closest to the bottom side of L. Hence, there is no unoccupied area directly below N_j, and moreover, the straight line of every w_j-slot is not below the extension, B_j, (of the bottom side of N_j) because otherwise, the algorithm would have chosen a different position for N_j.

Now, suppose for a contradiction that there is an unoccupied point, P, below B_i in S_l. We move upwards from P and reach a rectangle N_q. Note that we always reach a rectangle and that N_q is not equal to $N_{t'}^{\text{out}}$, because of the straight lines computed before. However, now there is an unoccupied point directly below a rectangle; a contradiction. □

Lemma 23. *Let N_i be an inside element. Then,*

(i) N_i can be placed as described in the algorithm, and

(ii) $\phi_i \leq \frac{5}{6}$.

Proof. (i) We will show that N_i's top side is below the bottom side of $N_{t'}^{\text{out}}$ (if $w_l = w_{t'}$) or below the top side of L (otherwise). Suppose for a contradiction that the top side of N_i is not below the top side of L; see Fig. 5.6. The proof works analogously in the other case. Consider the extension, B_i, of the bottom side of N_i.

The area below B_i is completely occupied (by Lemma 22), and hence, $f \leq w_l h_i - w_i h_i$, where f denotes the size of the total unoccupied area in S_l. Recall that $w_j/4 \leq h_j < w_j$, implying $n_j = w_j \cdot h_j \geq w_j^2/4$, for all j.

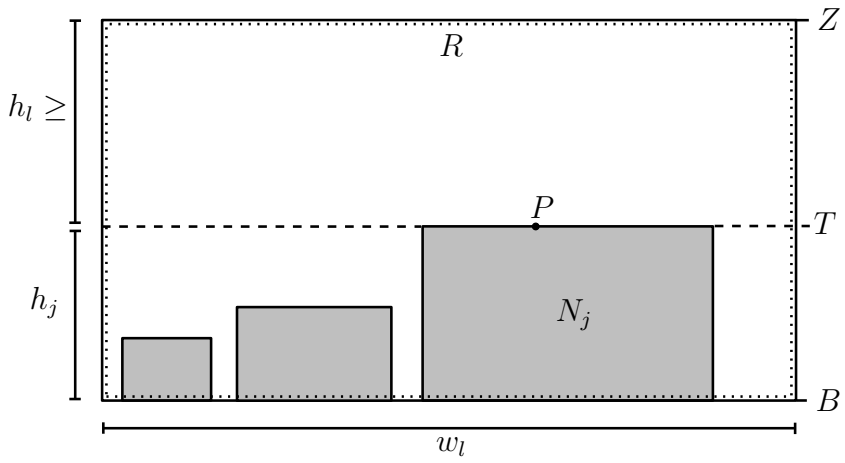

Figure 5.7: The rectangle R (dotted line). There area below B is completely occupied, the area above T and below Z is completely unoccupied. Z is either equal to the top side of L or, if $w_{t'} = w_l$, to the bottom side of $N_{t'}^{\text{out}}$.

Now, if $w_i \leq w_l/4$, then $f < w_l \cdot w_i \leq w_l^2/4 \leq n_l$, if $w_i = w_l/2$ then $f \leq (w_l - w_i)h_i = w_l/2 \cdot h_i < w_l/2 \cdot w_i = w_l/2 \cdot w_l/2 = w_l^2/4 \leq n_l$, or if $w_i = w_l$, then $f = 0$. In all three cases the magnitude of f contradicts the size of n_l, because by our choice, N_l fits completely into S_l.

(ii) N_i is a $w_i \times h_i$-rectangle with $w_i/4 \leq h_i < w_i$. Thus,

$$\begin{aligned} \phi_i &\leq \max_{w_i,h_i} \left\{ \frac{I[N_i]}{(w_i \cdot h_i)^{2.5}} \right\} \\ &= \max_{w_i,h_i} \left\{ \frac{\frac{1}{3}(h_i + w_i)(w_i \cdot h_i)^2}{(h_i \cdot w_i)^{2.5}} \right\} \\ &= \max_{w_i,h_i} \left\{ \frac{h_i + w_i}{3(h_i \cdot w_i)^{0.5}} \right\} . \end{aligned}$$

We set $h_i = \lambda \cdot w_i$, for $\lambda \in [\frac{1}{4}, 1]$, and we need to find

$$\max_{\lambda} \left\{ \frac{\lambda + 1}{3\sqrt{\lambda}} \right\} .$$

The first derivative of the function inside the brackets is equal to zero iff $\lambda = 1$, and the second derivative attains a value of $\frac{1}{6}$, for $\lambda = 1$. Thus, $\lambda = 1$ is a minimum. We obtain the maximal value of $\frac{5}{6}$ at the left boundary of the interval $[\frac{1}{4}, 1]$. Hence, $\phi_i \leq \frac{5}{6}$. □

The lid element is placed in the area that is left after placing the transition element and the inside elements. In particular, the shape of N_l does not need to be rectangular. However, we will show that N_l is placed in a rectangle that has bounded aspect ratio and is only a constant times larger than N_l.

Lemma 24. *Let N_l be a lid element. Then, $\phi_l \leq 3.5$.*

Proof. Consider the situation before N_l is placed. If $w_l = w_{t'}$, then $N_{t'}^{\text{out}}$ is a rectangle at the top of S_l. We denote by Z the bottom side of $N_{t'}^{\text{out}}$, if $w_l = w_{t'}$, or the intersection of the top side of L and S_l, otherwise; see Fig. 5.7.

Let F be the total unoccupied area in S_l, then the algorithm chooses $N_l = F$. F is clearly below Z. Now consider the highest occupied point, P, below Z in S_l. Let $P \in N_j$ and let B be the extension of the bottom side of N_j. By Lemma 22, the area below B in S_l is occupied. Thus, F lies completely above B. Moreover, let T be the extension of the top side of N_j. Because we chose P to be the highest occupied point in S_l below Z, the area above T and below Z is unoccupied. We consider two cases to bound the ϕ-value. If $w_j = w_l$: Then, N_l is a $w_l \times h_l$-rectangle, and hence, $\phi_l \leq 5/6$.

If $w_j \leq w_l/2$: First we analyze Case A of the algorithm. N_j lies inside the rectangle, R, formed by B, Z, the left and the right side of S_l. The height of R is at most $h_j + h_l$, because the area between T and Z is completely unoccupied, and N_l could be placed as a $w_l \times h_l$-rectangle if the area between B and T would be completely occupied. The width of R is w_l. There are the following restrictions on the side lengths:

$$\frac{w_j}{4} \leq h_j \leq w_j \ , \ \frac{w_l}{4} \leq h_l \leq w_l \ , \text{ and } \ 0 \leq w_j \leq \frac{w_l}{2} \tag{5.7}$$

Thus, $h_j \leq w_l/2$. The maximal distance inside R is $w_l + h_l + h_j \leq \frac{3}{2}w_l + h_l$. Moreover, $(w_l \cdot h_l)^2$ is twice the number of pairs of points in N_l. Hence, we get:

$$\phi_l \leq \max_{w_l, h_l} \left\{ \frac{(\frac{3}{2}w_l + h_l)(w_l \cdot h_l)^2}{(w_l \cdot h_l)^{2.5}} \right\} = \max_{w_l, h_l} \left\{ \frac{\frac{3}{2}w_l + h_l}{(w_l \cdot h_l)^{0.5}} \right\} \ .$$

We set $h_l = \lambda \cdot w_l$, for $\lambda \in [\frac{1}{4}, 1]$, and get

$$\phi_l \leq \max_{\lambda} \left\{ \frac{\frac{3}{2} + \lambda}{\sqrt{\lambda}} \right\} \ .$$

The first derivative of the right side is equal to zero iff $\lambda = \frac{3}{2}$. Hence, the function attains the maximum at the boundaries of the interval $[\frac{1}{4}, 1]$. We compute a value of 3.5 for $\lambda = \frac{1}{4}$ and a value of 2.5 for $\lambda = 1$. Thus, $\phi_l \leq 3.5$.

If N_l is placed according to Case B or C: Consider the line the algorithm moves down to construct the area for N_l. If the line reaches some rectangle that has been placed before, we can use the same analysis as in Case A. Otherwise, N_l is a $h_l \times w_l$-rectangle. Thus, $\phi_l \leq 3.5$ in all three cases. □

Remark Another upper bound for ϕ_l is obtained by computing $I[R \setminus N_j] \cdot (w_l \cdot h_l)^{-2.5}$ and then, maximizing this expression with respect to the constraints (5.7). If R's height set to $h_l + h_j$, this does not lead to a better ϕ-value than $\frac{61}{12} \approx 5.0833$.

Theorem 14. *For every $j \in \{1, 2, \ldots, k\}$, $\phi_j \leq 3.5$ holds, and the algorithm is a 5.3827-approximation.*

Proof. The ϕ-values are dominated by the lid elements. Every lid element has a ϕ-value less than or equal to 3.5. The value for an optimal shape is larger than 0.65024; compare [BBDF04]. □

5.4 Conclusion

We showed that optimal n-towns can be computed in time $O(n^{7.5})$. This is of both theoretical and practical interest, as it yields a method polynomial in n and extends the limits of the best-known solutions; however, there are still some ways how the result could be improved, in particular, it might be possible to lower the number of parameters in the dynamic program.

Our method is polynomial in n, but as the input size is $O(\log n)$, it is only a pseudo-polynomial algorithm. To obtain an algorithm polynomial in $O(\log n)$, one needs to describe the solution in polylogarithmic space. Besides the properties presented in Section 5.2.1, there are some additional properties of an optimal town, *e.g.,* there is always a "large" inner square and the height of two consecutive columns cannot differ "too much", but none of them yields a polylogarithmic description. Indeed, we are sceptical that one exists at all.

Moreover, we presented a 5.3827-approximation algorithm for the problem of packing cities into a unit square. This problem is of interest in the context of grid computing and malleable scheduling as it provides a method to select the number of processors, dedicated to a certain job, such that the communication cost is only a constant times larger than the optimal communication cost. Thus, one might decide on the number of processers used to process a job and on their arrangement separately. Currently, the communication cost is implicitly contained in the speed-up caused by increasing the number of processors working on the same job.

We believe that our algorithm can be improved. The bottleneck in our analysis is the choice of the lid element for which the exact form is not known. Hence, one approach might be to consider different shapes which tile the whole square and have a bounded ϕ-value.

Chapter 6

Conclusion

In this work, we discussed three different packing problems. Our focus was on the development of competitive algorithms (online problems) and approximation algorithms (offline problems).

We presented the Sequence Insertion Problem in Chapter 3. This problem differs from classical storage allocation problems because blocks always occupy contiguous subarrays and moves have to carry a block to a subarray that is disjoint from the current position of the block. In this setting, even inserting a single block into the array is an NP-hard problem. To achieve a minimum makespan, one can keep the blocks in sorted order inside the array. We presented a lower bound and a matching (up to a constant factor) upper bound of $O(n^2)$ for the problem of sorting n blocks inside an array. This forms the basis for the algorithms AlwaysSorted and DelayedSorted which achieve an optimal makespan. Moreover, we presented an algorithm (ClassSort) that uses only a constant number of block moves per insertion and deletion. All algorithms were evaluated experimentally.

In Chapter 4 we studied the strip packing problem with two additional constraints: Tetris constraint and gravity constraint. These constraints arise whenever the objects have actually be *packed* into strip, rather than *placed* in it. We improve the factor of 4 presented by Azar and Epstein [AE97], if the sequence consists of squares only. Moreover, their algorithm does not work in the presence of gravity; it can only handle the Tetris constraint. We presented two algorithms called BottomLeft and SlotAlgorithm. They have competitive factors of 3.5 and 2.6154, respectively. Moreover, we presented a lower bound of $\frac{5}{4}$ for any algorithm.

Two problems, both with the goal of minimizing the interior distances of point sets, are studied in Chapter 5. The first problem asks for a set of grid points of given cardinality and minimum average pairwise L_1 distances. We present different properties of such point sets and an algorithm that solves

the problem optimally in time $O(n^{7.5})$, for a set of cardinality n. For this problem, this is the first algorithm that runs in time polynomially bounded in n.

In the second—closely related—problem, one has to select shapes of given area inside a unit square. Again, the average pairwise L_1 distances have to be minimized. More precisely, our goal is to minimize the maximum of the average pairwise L_1 distances of the shapes. We presented a 5.3827-approximation algorithm for this problem.

List of Figures

Bibliography

[AE97] Yossi Azar and Leah Epstein. On two dimensional packing. *Journal of Algorithms*, 25(2):290–310, 1997.

[AFK+08] Josef Angermeier, Sándor P. Fekete, Tom Kamphans, Dirk Koch, Nils Schweer, Jürgen Teich, Christopher Tessars, and Jan C. van der Veen. No-break dynamic defragmentation of reconfigurable devices. In *Proceedings of the 18th International Conference on Field Programmable Logic and Applications (FPL'08)*, pages 113–118, 2008.

[AITT00] Yuichi Asahiro, Kazuo Iwama, Hisao Tamaki, and Takeshi Tokuyama. Greedily finding a dense subgraph. *Journal of Algorithms*, 34(2):203–221, 2000.

[AKK99] Sanjeev Arora, David R. Karger, and Marek Karpinski. Polynomial time approximation schemes for dense instances of NP-hard problems. *Journal of Computer and System Sciences*, 58(1):193–210, 1999.

[AL90] Arne Andersson and Tony W. Lai. Fast updating of well-balanced trees. In *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT'90)*, volume 447 of *Lecture Notes in Computer Science*, pages 111–121. Springer-Verlag, 1990.

[BBD+08] Michael A. Bender, David P. Bunde, Erik D. Demaine, Sándor P. Fekete, Vitus J. Leung, Henk Meijer, and Cynthia A. Phillips. Communication-aware processor allocation for supercomputers: Finding point sets of small average distance. *Algorithmica*, 50(2):279–298, 2008.

[BBDF04] Carl M. Bender, Michael A. Bender, Erik D. Demaine, and Sándor P. Fekete. What is the optimal shape of a city? *Journal of Physics A: Mathematical and General*, 37(1):147–159, 2004.

[BBK81] Brenda S. Baker, Donna J. Brown, and Howard P. Katseff. A 5/4 algorithm for two-dimensional packing. *Journal of Algorithms*, 2(4):348–368, 1981.

[BBK82] Donna J. Brown, Brenda S. Baker, and Howard P. Katseff. Lower bounds for on-line two-dimensional packing algorithms. *Acta Informatica*, 18(2):207–225, 1982.

[BC84] Brenda S. Baker and Edward G. Coffman, Jr. Insertion and compaction algorithms in sequentially allocated storage. *SIAM Journal on Computing*, 13(3):600–609, 1984.

[BCD+02] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02)*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164. Springer-Verlag, 2002.

[BCR80] Brenda S. Baker, Edward G. Coffman, Jr., and Ronald L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846–855, 1980.

[BCR01] Yair Bartal, Moses Charikar, and Danny Raz. Approximating min-sum k-clustering in metric spaces. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC'01)*, pages 11–20. ACM Press, 2001.

[BCW85] Brenda S. Baker, Edward G. Coffman, Jr., and Dan E. Willard. Algorithms for resolving conflicts in dynamic storage allocation. *Journal of the ACM*, 32(2):327–343, 1985.

[BDH+04] Ron Breukelaar, Erik D. Demaine, Susan Hohenberger, Hendrik Jan Hoogeboom, Walter A. Kosters, and David Liben-Nowell. Tetris is hard, even to approximate. *International Journal of Computational Geometry and Applications*, 14(1–2):41–68, 2004.

[BFKS09] Michael A. Bender, Sándor P. Fekete, Tom Kamphans, and Nils Schweer. Maintaining arrays of contiguous objects. In *Proceedings of 17th International Symposium on Fundamentals of Computation Theory (FCT'09)*, volume 5699 of *Lecture Notes Computer Science*, pages 14–25, 2009.

[BFM06] Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro. Insertion sort is $O(n \log n)$. *Theory of Computing Systems*, 39(3):391–397, 2006.

[BN95] Konrad Behnen and Georg Neuhaus. *Grundkurs Stochastik*. B.G.Teubner-Verlag, 3rd edition, 1995.

[Bro80] Allan G. Bromley. Memory fragmentation in buddy methods for dynamic storage allocation. *Acta Informatica*, 14(2):107–117, 1980.

[Bro96] Gerth Stølting Brodal. Worst-case efficient priority queues. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'96)*, pages 52–58. SIAM, 1996.

[Bru04] Peter Brucker. *Scheduling Algorithms*. Springer-Verlag, 2nd edition, 2004.

[BS83] Brenda S. Baker and Jerald S. Schwarz. Shelf algorithms for two-dimensional packing problems. *SIAM Journal on Computing*, 12(3):508–525, 1983.

[CDW02] Edward G. Coffman, Jr., Peter J. Downey, and Peter Winkler. Packing rectangles in a strip. *Acta Informatica*, 38(10):673–693, 2002.

[CFL90] Edward G. Coffman, Jr., Leopold Flatto, and Frank T. Leighton. First-fit storage of linear lists: Tight probabilistic bounds on wasted space. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'90)*, pages 272–279. SIAM, 1990.

[CGJT80] Edward G. Coffman, Jr., Michael R. Garey, David S. Johnson, and Robert E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826, 1980.

[CKS85] Edward G. Coffman, Jr., T. T. Kadota, and Larry A. Shepp. A stochastic model of fragmentation in dynamic storage allocation. *SIAM Journal on Computing*, 14(2):416–425, 1985.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CTB98] Mark E. Crovella, Murad S. Taqqu, and Azer Bestavros. Heavy-tailed probability distributions in the world wide web. In *A Practical Guide To Heavy Tails*, pages 3–26. Chapman & Hall, 1998.

[CW97] János Csirik and Gerhard J. Woeginger. Shelf algorithms for on-line strip packing. *Information Processing Letters*, 63(4):171–175, 1997.

[DD07] Erik D. Demaine and Martin L. Demaine. Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity. *Graphs and Combinatorics*, 23(1):195–208, 2007.

[DFR+09] Erik D. Demaine, Sándor P. Fekete, Günther Rote, Nils Schweer, Daria Schymura, and Mariano Zelke. Integer point sets minimizing average pairwise ℓ_1 distance: What is the optimal shape of a town? In *Proceedings of the 21st Canadian Conference on Computational Geometry (CCCG'09)*, pages 145–148, 2009.

[Die82] Paul F. Dietz. Maintaining order in a linked list. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC'82)*, pages 122–127. ACM Press, 1982.

[DS87] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC'87)*, pages 365–372. ACM Press, 1987.

[DSZ94] Paul F. Dietz, Joel I. Seiferas, and Ju Zhang. A tight lower bound for on-line monotonic list labeling. In *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory (SWAT '94)*, volume 824 of *Lecture Notes in Computer Science*, pages 131–142. Springer-Verlag, 1994.

[DZ90] Paul F. Dietz and Ju Zhang. Lower bounds for monotonic list labeling. In *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT'90)*, volume 447 of *Lecture Notes in Computer Science*, pages 173–180. Springer-Verlag, 1990.

[FG05] Gianni Franceschini and Viliam Geffert. An in-place sorting with $O(n \log n)$ comparisons and $O(n)$ moves. *Journal of the ACM*, 52(4):515–537, 2005.

[FKS09] Sándor P. Fekete, Tom Kamphans, and Nils Schweer. Online square packing. In *Proceedings of the 11th Algorithms and Data*

Structures Symposium (WADS'09), volume 5664 of *Lecture Notes in Computer Science*, pages 302–314. Springer-Verlag, 2009.

[FM03] Sándor P. Fekete and Henk Meijer. Maximum dispersion and geometric maximum weight cliques. *Algorithmica*, 38(3):501–511, 2003.

[FMB05] Sándor P. Fekete, Joseph S. B. Mitchell, and Karin Beurer. On the continuous Fermat-Weber problem. *Operations Research*, 53(1):61–76, 2005.

[FMW00] Sándor P. Fekete, Joseph S. B. Mitchell, and Karin Weinbrecht. On the continuous Weber and k-median problems. In *Proceedings of the 16th Annual Symposium on Computational Geometry (SoCG'00)*, pages 70–79, 2000.

[FS98] Sándor P. Fekete and Jörg Schepers. New classes of lower bounds for bin packing problems. In *Proceedings of the 6th International Conference on Integer Programming and Combinatorial Optimization (IPCO'98)*, volume 1412, pages 257–270. Springer-Verlag, 1998.

[FW98] Amos Fiat and Gerhard J. Woeginger, editors. *Online Algorithms: The State of the Art*. Springer, 1998.

[GBH98] Nili Guttmann-Beck and Refael Hassin. Approximation algorithms for minimum sum p-clustering. *Discrete Applied Mathematics*, 89(1–3):125–142, 1998.

[Ger96] Jordan Gergov. Approximation algorithms for dynamic storage allocations. In *Proceedings of the 4th Annual European Symposium on Algorithms (ESA'96)*, volume 1136 of *Lecture Notes in Computer Science*, pages 52–61. Springer-Verlag, 1996.

[GF93] Gabor Galambos and J. B. G. Frenk. A simple proof of Liang's lower bound for online bin packing and the extension to the parametric case. *Discrete Applied Mathematics*, 41(2):173–178, 1993.

[GJ79] Michael R. Garey and David S. Johnson. *Computer and Intractability — A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[GO97] Jacob E. Goodman and Joseph O'Rourke, editors. *Handbook of Discrete and Computational Geometry*. CRC Press, 1997.

[Gra69] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[GW95] Gabor Galambos and Gerhard J. Woeginger. Online bin packing - a restricted survey. *Mathematical Methods of Operations Research*, 42(1):25–45, 1995.

[Hin75] James A. Hinds. An algorithm for locating adjacent storage blocks in the buddy system. *Communications of the ACM*, 18(4):221–222, 1975.

[Hir73] Daniel S. Hirschberg. A class of dynamic memory allocation algorithms. *Communications of the ACM*, 16(10):615–618, 1973.

[HIYZ07] Xin Han, Kazuo Iwama, Deshi Ye, and Guochuan Zhang. Strip packing vs. bin packing. In *Proceedings of the 3rd International Conference on Algorithmic Aspects in Information and Management (AAIM'07)*, volume 4508 of *Lecture Notes in Computer Science*, pages 358–367. Springer-Verlag, 2007.

[HLS09] R. Hassin, A. Levin, and M. Sviridenko. Approximating minimum quadratic assignment problems. *To appear*, 2009.

[Hoc97] Dorit Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 1997.

[HRT97] Refael Hassin, Shlomi Rubinstein, and Arie Tamir. Approximation algorithms for maximum dispersion. *Operations Research Letters.*, 21(3):133–137, 1997.

[IKR81] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431. Springer-Verlag, 1981.

[Ind99] Piotr Indyk. A sublinear time approximation scheme for clustering in metric spaces. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS'99)*, pages 154–159, 1999.

[Irl93] Gordon Irlam. Unix file size survey - 1993. URL http://www.gordoni.com/ufs93.html, 1993.

[KMN+97] Sven O. Krumke, Madhav V. Marathe, Hartmut Noltemeier, Venkatesh Radhakrishnan, S. S. Ravi, and Daniel J. Rosenkrantz. Compact location problems. *Theoretical Computer Science*, 181(2):379–404, 1997.

[KMW75] Richard M. Karp, A. C. McKellar, and C. K. Wong. Near-optimal solutions to a 2-dimensional placement problem. *SIAM Journal on Computing*, 4(3):271–286, 1975.

[Kno65] Kenneth C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–625, 1965.

[Knu97a] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, 3rd edition, 1997.

[Knu97b] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 2nd edition, 1997.

[KP93] Guy Kortsarz and David Peleg. On choosing a dense subgraph. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Comptuer Science (FOCS'93)*, pages 692–703. IEEE Computer Society Press, 1993.

[KR96] Claire Kenyon and Eric Rémila. Approximate strip packing. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science, (FOCS'96)*, pages 31–36. IEEE Computer Society Press, 1996.

[LAB+02] Vitus J. Leung, Esther M. Arkin, Michael A. Bender, David Bunde, Jeanette Johnston, Alok Lal, Joseph S. B. Mitchell, Cynthia Phillips, and Steven S. Seiden. Processor allocation on Cplant: Achieving general processor locality using one-dimensional allocation strategies. In *Proceedings of the 4th IEEE International Conference on Cluster Computing (CLUSTER'02)*, pages 296–304, 2002.

[LAB+07] Eliane M. Loiola, Nair M. M. de Abreu, Paulo O. Boaventura-Netto, Peter Hahn, and Tania Querido. A survey for the quadratic assignment problem. *European Journal of Operational Research*, 176(2):657–690, 2007.

[LMM02] Andrea Lodi, Silvano Martello, and Michele Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, 2002.

[LNO96] Michael G. Luby, Joseph Naor, and Ariel Orda. Tight bounds for dynamic storage allocation. *SIAM Journal on Discrete Mathematics*, 9(1):155–166, 1996.

[LTW+90] Joseph Y.-T. Leung, Tommy W. Tam, C. S. Wong, Gilbert H. Young, and Francis Y. L. Chin. Packing squares into a square. *Journal of Parallel and Distributed Computing*, 10(3):271–275, 1990.

[Mey07] Uwe Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays*. Springer-Verlag, 2nd edition, 2007.

[MG78] Robert C. Melville and David Gries. Sorting and searching using controlled density arrays. Technical report, 1978.

[ML96] Jens Mache and Virginia Lo. Dispersal metrics for non-contiguous processor allocation. Technical Report CIS-TR-96-13, University of Oregon, 1996.

[ML97] Jens Mache and Virginia Lo. The effects of dispersal on message-passing contention in processor allocation strategies. In *Proceedings of the 3rd Joint Conference on Information Sciences (JCIS'97)*, pages 223–226, 1997.

[MR96] J. I. Munro and Venkatesh Raman. Fast stable in-place sorting with $O(n)$ data moves. *Algorithmica*, 16(2):151–160, 1996.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Rei06] Joachim Reichel. *Combinatorial approaches for the trunk packing problem*. PhD thesis, Universität des Saarlandes, 2006.

[Rob77] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *Computer Journal*, 20(3):242–244, 1977.

[RRT94] S. S. Ravi, D. J. Rosenkrantz, and G. K. Tayi. Heuristic and special case algorithms for dispersion problems. *Operations Research*, 42(2):299–310, 1994.

[Sch94] Ingo Schiermeyer. Reverse-fit: A 2-optimal algorithm for packing rectangles. In *Proceedings of the 2nd Annual European Symposium on Algorithms (ESA'94)*, volume 855 of *Lecture Notes in Computer Science*, pages 290–299. Springer-Verlag, 1994.

[Sei02] Steven S. Seiden. On the online bin packing problem. *Journal of the ACM*, 49(5):640–671, 2002.

[SG76] Sartaj Sahni and Teofilo Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23(3):555–565, 1976.

[Sle80] Daniel D. Sleator. A 2.5 times optimal algorithm for packing in two dimensions. *Information Processing Letters*, 10(1):37–40, 1980.

[SP74] Kenneth K. Shen and James L. Peterson. A weighted buddy method for dynamic storage allocation. *Communications of the ACM*, 17(10):558–562, 1974.

[SPG94] Abraham Silberschatz, James L. Peterson, and Peter B. Galvin. *Operating Systems Concepts*. Addison-Wesley, 4th edition, 1994.

[Ste97] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26(2):401–409, 1997.

[Ste08] Rob van Stee. Combinatorial algorithms for packing and scheduling problems, 2008. Habilitationsschrift, University of Karlsruhe.

[Tsa84] Athanasios K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–112, 1984.

[Vli92] André van Vliet. An improved lower bound for online bin packing algorithms. *Information Processing Letters*, 43(5):277–284, 1992.

[Wil92] Dan E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, 1992.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[YHZ09] Deshi Ye, Xin Han, and Guochuan Zhang. A note on online strip packing. *Journal of Combinatorial Optimization*, 17(4):417–423, 2009.

[Zha93] Ju Zhang. *Density control and on-line labeling problems.* PhD thesis, University of Rochester, 1993.

[Zha04] Hu Zhang. Approximation algorithms for min-max resource sharing and malleable tasks scheduling, 2004. PhD thesis, University of Kiel.

Index